A Design Space and its Patterns: Modelling 2phase Asynchronous Pipelines

Graham Birtwistle[1]
and Kenneth S. Stevens[2]

[1] Sheffield University, Yorkshire, UK
graham@dcs.sheffield.ac.uk
[2] University of Utah, Salt Lake City, USA
kstevens@ece.utah.edu

Abstract

We present a systematic way of studying state machine based design spaces and apply it to the study of asynchronous pipelines. Starting with the specification of the most concurrent behaviour as a state machine, all possible valid smaller designs may be generated by systematically removing structured patterns of output states ($\mathcal{L}$ cuts) and input states ($\mathcal{R}$ cuts). Taking the cartesian product of cuts $\mathcal{L}\times\mathcal{R}$ yields the complete design space which may then be partitioned according to well understood design styles. In this paper we extend previous results by studying mixed asynchronous pipelines of which homogeneous behaviours form a subset. The approach is presented using the much smaller 2phase setting (3×6) but the insights and structures revealed carry over to full 4phase designs (35×140). We present a complete overview of mixed 2phase linear pipeline behaviours; show how their structuring $\mathcal{L}$ cuts and $\mathcal{R}$ cuts relate; characterise the behaviours of linear pipelines in terms of these cuts for any depth; and show how the much larger $\mathcal{R}$ mixed behaviour patterns can be calculated from knowledge of the $\mathcal{L}$ behaviour patterns. Applications of the theory cover mixed linear pipeline and ring behaviours and the automatic generation of quality circuits from our specifications.

1 Setting and approach

This work arose from our long standing interest in designing, specifying and building asynchronous microprocessors [2, 4, 5, 7, 21, 38, 39]. In the first stage of design development, our practice is to concentrate solely upon control signals and the ways in which they can interleave. This enables us to check that each subsystem and its compositions work together harmoniously (are live, deadlock free, preserve essential cyclic properties, ...) before extending the model towards data movements and calculations.

The computational core of a microprocessor lies in its pipelined datapath. (At the control signal level, combinational circuits minimise down and can be considered mere delays.) When we experimented with structured pipelines of differing widths and depths, they minimised down to an observationally equivalent linear pipeline structure of the same depth, but usually one not composed from the controller used in its design. This aroused our curiosity.

Our notation of choice has been CCS, Milner's Calculus of Communicating Systems [1, 32]. CCS is a system description language based upon communicating agents (state machines). Milner noticed that concurrent processes have an algebraic structure: given processes P and Q, we can construct new processes combining P and Q sequentially (example in Section 2.2) or in parallel (example in Section 2.3). The resulting composition will be a new process (in our case, system of circuits) whose behaviour depends upon those of P and of Q and of their combining operator. Further CCS provides just one inter-process communication mechanism which corresponds directly to the asynchronous circuit handshake. CCS has a number of pertinent attributes: it models arbitrary delays and interleaving behaviours which makes it straightforward to capture the signal level behaviour of asynchronous circuits and systems; and it has a simple and well understood formal semantics to support reasoning about designs, their properties,

and their equivalences. Additionally CCS has a reliable public domain tool support, the CWB (Concurrency Workbench [33]), with built-in minimisation to the smallest equivalent state machine and an implementation of the very powerful modal-μ property checking language—see Stirling's sterling account [41, Chapter 6, pages 141-153].

1.1 Previous Work and its Approach

Our previous work [6, 8, 34] has been concerned with mainly 4phase asynchronous homogeneous linear and structured parallel pipelines. The asynchronous community has made great efforts to present circuits clearly [9, 14, 16, 18, 19, 22, 23, 24, 25, 26, 27, 35, 36, 42, 44, 46]. This body of work enabled us to model real practical designs rather than experiment with a few idealised ones and kept us grounded. Importantly, the corpus was large to guide our research directions.
Initial Survey. It is common practice within the community to specify designs graphically (STG [13, 37, 45]) or algebraically (CSP [11, 12, 26, 28, 29, 30, 31]) at the signal level—our chosen level of abstraction. Some forty published 4phase designs were translated from their original presentations into CCS. STG specifications include internal state variables which assist the favoured CAD tool Petrify [15] to target a specific implementation by delaying outgoing signals until an internal state has been reached. We first mapped all these signals into CCS and then, by simply hiding any explicit mention of an STG internal variable name, retained its inherent constraints as extra arbitrary length internal delays. In this way, our CCS model characterizes the signal interleaving possibilities relevant to pipelining. To emphasize this distinction, we call our minimised CCS state machine descriptions *shapes*. Shapes operate at the same level of abstraction as STGs and our specifications lead directly to circuits in silicon [34]. Brunvand, Lines and Martin have used CSP equally abstractly to generate silicon [12, 26, 31, 44].
Generalisation. The forty 4phase designs surveyed in the published literature gave rise to just 18 distinct shapes. Since the behaviour of each and every shape is expressed solely in terms of external control signals, shape behaviours could then be compared, contrasted, and ordered. From comparison of these shapes, we noted a mathematically ideal shape, max_1, from which all published designs could be defined by removing states. We claim that max_1 is the most concurrent possible shape. In support of this claim, the union of published shapes is precisely max_1. This led to generating all possible subshapes by systematically cutting away states from max_1. This is easiest to carry out in an algebraic notation—hence the aptness of CCS.

1.2 Intuition: MAX and its cutaways.

Consider the FSM *MAX* shown at the top of Figure 1. *MAX* has 20 states arranged in 5 rows and 4 columns. The initial state is marked **START** and the terminal state **DONE**. Informally, the protocol is to take us from the start state to the done state by making only forward $\rightarrow$ or downward $\downarrow$ moves. To save clutter we have omitted arrowheads and labels: horizontal arcs run

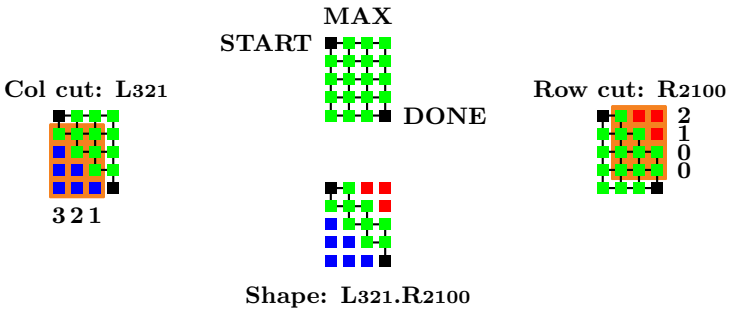


Figure 1: MAX and its cutaways

to the right; vertical arrows downwards. The extreme routes are **path DA**: down 4 arcs, then across 3 arcs and **path AD**: across 3 arcs, then down 4 arcs. The middle section of Figure 1 divides the cut possibilities into two regions shown with shaded backgrounds. On the left, we have a 4×3 region with the specific column cut $\mathbf{L}_{321}$ displayed. Starting at the bottom of each column, $\mathbf{L}_{321}$ cuts away *upwards*: 3 states from column 1, 2 states from column 2, and 1 state from column 3—plus of course their connecting arcs. Clearly we should never cut away the start state or the done state. On the right, we have a 4×3 region with the specific row cut $\mathbf{R}_{2100}$ displayed. Starting at the right hand end of each row, $\mathbf{R}_{2100}$ cuts away *leftwards* 2 states from row 1, 1 from row 2, and 0 from rows 3 and 4. The typical sub-design from *MAX* will have cutaways on the left and on the right, for example $\mathbf{L}_{321}.\mathbf{R}_{2100}$ at the bottom of Figure 1.

Liveness. As illustrated in Figure 2, the choice of indices in $\mathbf{L}_{abc}$ or $\mathbf{R}_{wxyz}$ is not entirely free. In this figure, the START/DONE states are marked ■, cut states by ■ on the left and ■ on the right, and 'dead ends' by ●.

Figure 2: Non-valid cuts of classes 2 and 3

1. **Range:** clearly 0 ≤ a,b,c ≤ 4; and 0 ≤ w,x,y,z ≤ 3.
2. **No dead-end states:** as would be caused by cuts $\mathbf{L}_{010}$ and $\mathbf{R}_{0120}$ on the left in Figure 2. For $\mathbf{L}_{010}$, the state marked ● in row 5 and column 1 is a dead-end—reachable but isolated. The appropriate cut choice here is $\mathbf{L}_{110}$. For $\mathbf{R}_{0120}$ the 3 states marked ● are also dead ends. The appropriate cut choice here is $\mathbf{R}_{2220}$.
   Constraints to remove such options are: $\mathbf{L}_{abc}$: a ≥ b ≥ c **and** $\mathbf{R}_{wxyz}$: w ≥ x ≥ y ≥ z
3. **Liveness:** Each shape $\mathbf{L}_{abc}.\mathbf{R}_{wxyz}$ represents a viable sub-design provided that it is *live*. If its L- and R-cuts abut or *a fortiori* overlap then the shape will not be live—it will not be able to fulfill the protocol and move from the start state to the done state, as with $\mathbf{L}_{220}.\mathbf{R}_{2222}$ on the right in Figure 2 where the cuts abut. Liveness can be calculated from the L.cut and R.cut indices.

Design space. The sets of all allowable L.cuts, $\mathcal{L}$ = { $\mathbf{L}_{000}$, $\mathbf{L}_{001}$, $\mathbf{L}_{002}$,..., $\mathbf{L}_{444}$ } and the set of all allowable R.cuts, $\mathcal{R}$ = { $\mathbf{R}_{0000}$, $\mathbf{R}_{0001}$, $\mathbf{R}_{0002}$, ..., $\mathbf{R}_{3333}$ }, form regular and elegant lattices and enable us to compare and order the shapes (sub-designs) of *MAX*. Applying all combinations of left and right cuts $\mathcal{L}\times\mathcal{R}$ over *MAX* yields its complete space design space.

L000 → L444		$\mathcal{L}\times\mathcal{R}$
L000.R0000	L444.R0000	R 0 0 0 0 → R 3 3 3 3
DESIGN SPACE		
L000.R3333	L444.R3333	

Figure 3: *MAX* design space

This permits exhaustive examination of circuit design possibilities and their pipeline behaviours.

1.3 Contributions

In this paper we apply these ideas to studying and predicting the behaviours of mixed pipelines. Homogeneous results drop out as a subset in this study. Mixed pipelines enjoy a huge increase in variety over homogeneous pipelines, but this very generality has uncovered unifying results with practical applications. In particular, our previous R.cut sets [8, 34] are closed under composition for homogeneous pipelines, but not for mixed. A simple modification has led to two key discoveries:

1. A unique notation for pipelines of any depth in terms of the closed set of shape cutaways.

2. The relation between the L.cut set and the R.cut set. We already had a neat lattice for L.cuts as a planar wedge. We can now restructure the closed R.cuts into (2 for 2phase, 4 for 4phase) related planar wedges each of which is isomorphic to the L wedge. This has a practical application when we come to predict pipeline behaviours, since experiments show that if we know how a L.cut behaves we can calculate how a related R.cut behaves.

The full 4phase design space has 35 L.cuts and 140 R.cuts; its untimed sub-space (delay insensitive and speed independent shapes only) has 10 L.cuts and 20 R.cuts. Evaluating their design spaces for just pipelines of depth 2 would entail 24.01 million and 160 thousand experiments respectively as against just 324 for 2phase. Accordingly we present exhaustive practical results for the 3×6 2phase design style. Work in progress confirms that our 2phase insights and structures carry over into the above 4phase design spaces (see example in Section 5.4).

1.4 Structure of the paper

In the remaining sections in this paper: Section 2 introduce the CCS notation via examples leading to the model of data transmission which underlies pipeline behaviours. Section 3 explains the design and implementation of Furber's classic 2phase pipeline stage (which we call max_1) using simple building blocks expressed in CCS. It also covers pipeline models constructed from max_1 and discusses their specification. Section 4 covers the structure of cuts, their lattices, how they are related, and how they define the family of subdesigns from max_1. Section 5 presents new results on mixed pipelines, uses lattices to uncover their pipeline structures and pipelined behaviours; and predicts the behaviours of mixed pipelines and rings. Section 6 gives applications to circuit design. Section 7 is an overview and summary.

2 CCS as a System Description Language

CCS describes objects (agents, circuits, processes) by defining the states they can occupy and the actions that cause them to move from state and to another state or back to the same state. In this section we give an introduction to defining individual objects in CCS, how objects communicate via handshakes, and give a simple model of 2phase bundled data transmission protocol as a lead in to asynchronous modelling.

2.1 Individual objects

0: The simplest object in CCS is **0** which can do nothing; it cannot receive or send signals, nor has it any local actions. It is said to be *deadlocked*.
Prefixing: Sequential objects can be built from **0** by *prefixing actions* which are executed in order. For example, $Match1 = strike.\overline{burn}.\mathbf{0}$ describes an object that be struck, then burns, whereupon it deadlocks (is spent). The separating dot **.** indicates an arbitrary delay and may be read as *then some time later*. Except for the deadlocked state, **0**, states always start with

upper case letters, e.g. *Match1*. Actions always start with lower case letters. We distinguish between input actions (e.g. *strike*) and output actions (e.g. $\overline{burn}$) which are over-barred.
Choice: Not all objects are sequential. Many have choices of action stream. For example, $Match2 = strike.(\overline{burn}.\mathbf{0} + \mathsf{fail}.Match2 + \mathsf{dead}.\mathbf{0})$ describes a match that after being struck has 3 distinct behavioural options: it may burn and become spent *or* it may fail to light whereupon it may choose to repeat the action repertoire from the beginning, *or* (perhaps the match head has dropped off) is thrown away.

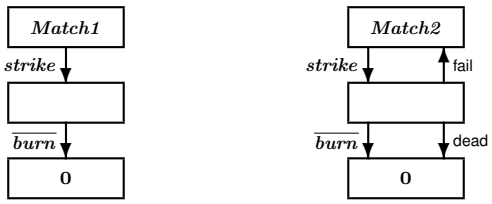


Figure 4: State machines for *Match1* and *Match2*

Figure 4 shows state machine descriptions of the linear *Match1* and the cyclic *Match2*. Notice the tight correspondences with their CCS specifications. We have made fail and dead local *trace* actions. Trace variables are useful documentation aids at key points and can be invaluable when property checking. We always show them in serif font.

2.2 Sequential Composition

Suppose we are given two objects, *FST* and *SND*, each of which upon receiving a request, carries out a local task, emits an acknowledgement and is then ready to repeat its cycle. We wish to allow *FST* and *SND* to cooperate in series: having completed a task1, *FST* hands over for *SND* to finish off by carrying out a task2. This communication is arranged by connecting the output acknowledgement of *FST* to the input request of *SND* as in Figure 5.

$$\boldsymbol{FST = rF.\mathsf{task1}.\overline{aF}.FST} \qquad \boldsymbol{SND = rS.\mathsf{task2}.\overline{aS}.SND}$$

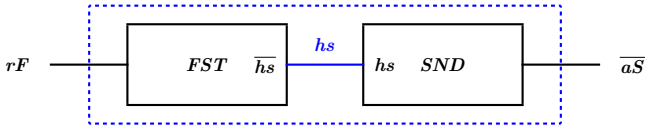


Figure 5: *FS*: handshake communication

For two objects to communicate via a direct *handshake* they must:

1. Agree on a common name for their communication line, here hs
2. Rename the sender's action to match the line name, that is in *FST* $\overline{aF} \to \overline{hs}$
3. Rename the receiver's action to match the line name, that is in *SND* $rS \to hs$

Table 1 defines *FS* as the *parallel composition* of *FST* and *SND*. It first lists the constituent objects, parenthesised and with separator |, here $(FST \mid SND)$; followed by its curly bracketed handshake lines (in general they are separated by commas, but here) $\setminus \{$ hs $\}$. This specification has been *embellished* by ⇳ arrows. They are not part of formal CCS—here they highlight the handshake.

$$
\begin{array}{rcl}
FST & = & rF \;.\; \mathsf{task1} \;.\; \overline{hs} \;.\; FST \\
 & & \Updownarrow \\
SND & = & hs \;.\; \mathsf{task2} \;.\; \overline{aS} \;.\; SND \\
FS & = & (\; FST \mid SND \;) \setminus \{\; \mathbf{hs} \;\}
\end{array}
$$

Table 1: Specification of *FS*

2.2.1 Communicating via handshakes

To explain the mechanics of handshaking, we unroll each of the definitions of *FST* and *SND* in Table 1 up to their third handshake:

$$
\begin{array}{lccccccccccccccc}
FST: & rF & . & \mathsf{task1} & . & \overline{hs} & . & rF & . & \mathsf{task1} & . & \overline{hs} & . & rF & . & \mathsf{task1} & . & \overline{hs} & . & \ldots\ldots \\
 & & & & & \Updownarrow & & & & & & \Updownarrow & & & & & & \Updownarrow & & \\
SND: & & & & & hs & . & \mathsf{task2} & . & \overline{aS} & . & hs & . & \mathsf{task2} & . & \overline{aS} & . & hs & . & \ldots\ldots
\end{array}
$$

Each thread in the composition is allowed to carry out its sequence of non-handshaking actions freely up to, but not past, a $\overline{hs}$ or hs signal. Thus *SND* is initially blocked with its handshake action hs uncovered, but *FST* may execute its initial rF and then its task1 action until it uncovers the companion handshake action $\overline{hs}$. The situation is now:

$$
\begin{array}{lccccccccccccccc}
FST: & \overline{hs} & . & rF & . & \mathsf{task1} & . & \overline{hs} & . & rF & . & \mathsf{task1} & . & \overline{hs} & . & \ldots\ldots \\
 & \Updownarrow & & & & & & \Updownarrow & . & & & & & \Updownarrow & & \\
SND: & hs & . & \mathsf{task2} & . & \overline{aS} & . & hs & . & \mathsf{task2} & . & \overline{aS} & . & hs & . & \ldots\ldots
\end{array}
$$

For handshakes and only handshakes, both cooperating partners move on through their own handshake action at the same time. This special *internal* or not-observable handshake action is designated a τ-action. After the τ, *FST* is free to start a fresh cycle and *SND* may proceed to get on with its local task2 action. These actions are interleaved until the next handshake forces the second rendezvous.

$$
\begin{array}{lccccccccccccc}
FST: & rF & . & \mathsf{task1} & . & \overline{hs} & . & rF & . & \mathsf{task1} & . & \overline{hs} & . & \ldots\ldots \\
 & & & & & \Updownarrow & . & & & & & \Updownarrow & & \\
SND: & \mathsf{task2} & . & \overline{aS} & . & hs & . & \mathsf{task2} & . & \overline{aS} & . & hs & . & \ldots\ldots
\end{array}
$$

Here is an equivalent specification for the *FS* of Table 1 given monolithically—expressed in terms of just . and +.

$$
\begin{array}{lcl}
Y11 & = & rF.Y12 \\
Y12 & = & \mathsf{task1}.Y13 \\
Y13 & = & \tau.Y14 \\
Y14 & = & rF.Y15 \;+\; \mathsf{task2}.Y24 \\
Y15 & = & \mathsf{task1}.Y16 \;+\; \mathsf{task2}.Y25 \\
Y16 & = & \mathsf{task2}.Y26
\end{array}
\qquad
\begin{array}{lcl}
Y24 & = & rF.Y25 \;+\; \overline{aS}.Y11 \\
Y25 & = & \mathsf{task1}.Y26 \;+\; \overline{aS}.Y12 \\
Y26 & = & \overline{aS}.Y13
\end{array}
$$

Table 2: Monolithic specification of *FS*

After the τ handshake, both *FST* and *SND* can move on in their own good time. One extreme possibility is that *FST* makes the next two moves, rF and then task1, but it is then stuck awaiting its handshake partner. There it remains until *SND* has completed its task2 and $\overline{aS}$ actions. Only then can the next handshake take place. The other extreme possibility is that *FST* is stuck, and *SND* makes its two moves before *FST* makes an rF move. The specification covers all possible interleaving patterns between these two extremes. This is easier to see from the monolithic specification 2.

2.2.2 Minimisation

The state diagram for a *minimised* specification of *FS* is:

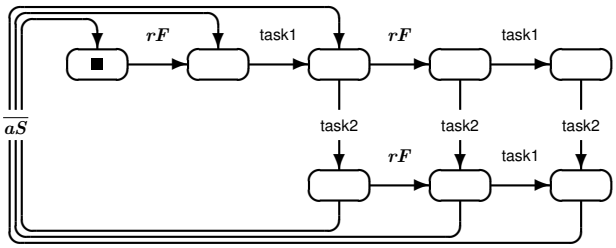


Figure 6: State graph of minimised *FS*

This is the smallest (in state size) definition of *FS* which is observably equivalent to the specification given in Table 1. Compared with the monolithic specification in Table 2, the non-observable τ action is deleted. The definition of state *Y13* is deleted and remaining references to it are updated to *Y14*.

Coda. For all but the simplest machines, it usually pays to define an object in terms of its interfaces (as we did here with *FST* and *SND*) and then constrain them. This composed specification will be the easiest to reason about, and once satisfied, the CWB will produce its minimised form automatically.

We have now given all the syntax and (informal) semantics for CCS that we need: the operators . + |, hiding, and handshaking. This enables us to deal with asynchronous hardware communication signals in appropriate detail. As with all state machine based descriptions, incorporating data would, of course, entail an exponential growth in state size.

2.3 2phase Bundled Data Transmission

We now present a simple model consisting of an output *O* and an input *I* cooperating via the 2phase protocol. It forms the core of how data gets transmitted down an asynchronous pipeline (see next section) and is an important first step in understanding the much used 2phase bundled data protocol.

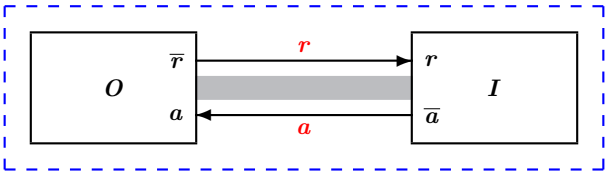


Figure 7: Transmitting bundled data between processes

Suppose that *O* and *I* are connected by a bus down which data values are transmitted in sequence from *O* to *I*. r/a are request/acknowledge communication wires respectively which ensure the safety of each transmission. Once *O* has loaded the next data value onto the bus, it sends a signal on line r. On receipt of which, *I* understands that a fresh data value is available and copies it locally from the bus. Once this data capture is complete, *I* signals on a to inform *O* that it can start the next transaction and safely overwrite the bus with the next data value. Using `pD` and `gD` as trace variables, the CCS description of this system is:

$$\begin{array}{rcl} O & = & \mathsf{pD} \;.\; \overline{r} \;.\; a \;.\; O \\ I & = & r \;.\; \mathsf{gD} \;.\; \overline{a} \;.\; I \\ OI & = & (\; O \mid I \;) \setminus \{ \mathtt{r}, \mathtt{a} \} \end{array}$$

Table 3: Specification of *OI*

Commentary on Table 3

pD: O puts fresh data on the bus linking O to I.
$\overline{r}\Leftrightarrow r$: When the data is stable, O sends a request signal on communication line r to I. O now passively awaits an acknowledgement from I.
gD: On receipt of the request signal on r, I reads the data from the bus.
$\overline{a}\Leftrightarrow a$: Once the data is captured, I sends an acknowledgement signal on line a back to O and passively awaits the next request

On receipt of a, O knows that the current data value has been safely passed to I. O may now actively prepare the next data value and place it on the bus. In Table 4 we unroll the specification of SR through the specific transaction.

	Transaction 1				•	Transaction 2				•
O =	pD . $\overline{r}\uparrow$.		$a\uparrow$		•	pD . $\overline{r}\downarrow$.		$a\downarrow$		• ....
	⇕		⇕			⇕		⇕		
I =		$r\uparrow$. gD .	$\overline{a}\uparrow$		•		$r\downarrow$. gD .	$\overline{a}\downarrow$		• ....

Table 4: Unrolling the specification of *OI* through 2 iterations

It is easy to show formally that OI is observationally equivalent to both specifications below:

$$\begin{array}{rcl} OI2 & = & \mathsf{pD} \;.\; \tau \;.\; \mathsf{gD} \;.\; \tau \;.\; OI2 \\ OI3 & = & \mathsf{pD} \;.\; \mathsf{gD} \;.\; OI3 \end{array}$$

which confirm that pD and gD alternate. Reading $\prec$ as *precedes*, $\mathsf{pD}_k \prec \mathsf{gD}_k$ means that I always reads fresh data; $\mathsf{gD}_k \prec \mathsf{pD}_{k+1}$ means that O cannot overwrite unread data. So the protocol is data independent and safe. As a final remark, transaction phases are identical in the 2phase protocol, so there is no need to indicate parity by $\uparrow$ and $\downarrow$.

2.3.1 4phase Protocol

In the equivalent 4phase version of this protocol, each signal goes up *and* down once per transaction. Hence its alternative name: Return To Zero, or RTZ.

	Transaction 1							•
O =	pD1 . $\overline{r}\uparrow$.		$a\uparrow$.		$\overline{r}\downarrow$.		$a\downarrow$	• ....
	⇕		⇕		⇕		⇕	
I =		$r\uparrow$. gD1 .	$\overline{a}\uparrow$.		$r\downarrow$.		$\overline{a}\downarrow$	• ....

At first sight this seems a waste of energy and time compared to 2phase. But 4phase hardware circuits may be simpler than 2phase and the down phase of one transaction may overlap with the up phase of its successor. The increased variety presents greater opportunities and greater challenges for engineers. It also gives rise to surprisingly larger design spaces.

2.4 Aptness of CCS for Circuits at the Control Signal Level

In later sections we will apply our algebraic approach to specifying the compositional properties of both homogeneous and mixed linear asynchronous pipeline and ring structures. We will explain how to describe basic asynchronous circuits as objects (state machines), the standard handshake method of synchronising circuits, how to compose cooperating circuits into systems, all the while using the 2phase design communication protocol and modelling in CCS. Which begs the question 'How well does CCS capture asynchronous hardware?'.

Circuit description. It is standard practice to describe a circuit as an asynchronous state machine. *This is exactly the CCS model for describing an object.*

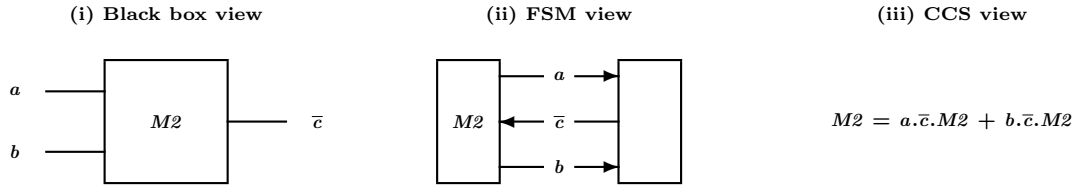


Figure 8: Black box, FSM and CCS views of M2

Signals as actions. Cooperating circuits must agree upon common communication lines; one sending and one receiving per line. Signals are simply changes in voltage level. We map signals to their line names: $\overline{x}/x$ will be interpreted as signals sent/received on line x. *This is exactly the CCS model for communicating via handshakes.*

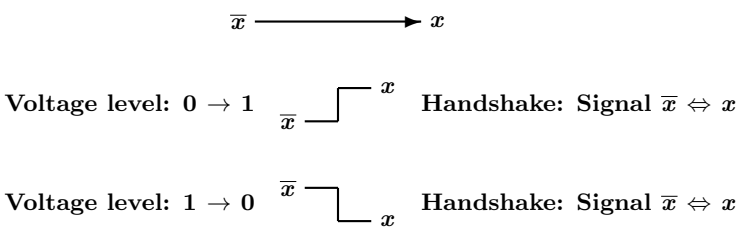
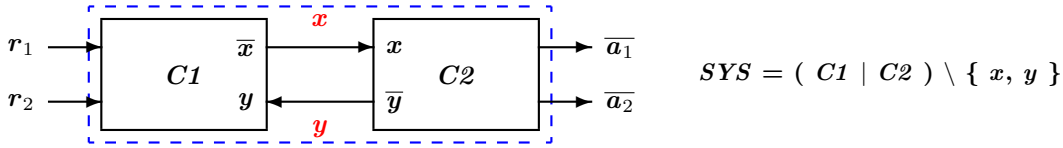


Figure 9: 2phase signal = change in voltage level, either up or down

System description. Hardware systems are built by wiring together the communication lines of their constituent circuits appropriately. *This is exactly the CCS model for building systems hierarchically.*



The handshakes are unobservable internal actions leaving the specification of *SYS* expressed solely in terms of the interleaving possibilities for its inputs and outputs: r_1, r_2, $\overline{a_1}$ and $\overline{a_2}$.

2phase protocol. In asynchronous systems, there are no coordinating clocks. In 2phase design, each circuit is passive until awoken by a request signal (or possibly several). When it has finished its current task, it will send a completion signal (or possibly several) and then fall passive again awaiting its next request. *This is modelled in CCS using the req/ack protocol as in the 2phase data transmission example of Section 2.3.*

3 Modelling asynchronous pipelines in CCS

In this section we describe Furber's classic implementation of a 2phase pipeline stage [20] and use this description to specify a suitable abstraction in CCS. Under common assumptions, we argue that this is the most concurrent behaviour achievable. We then recount experiments with this stage when pipelined and report their resulting structures and patterns. These patterns form the basis for specifying pipelines in the next sections.

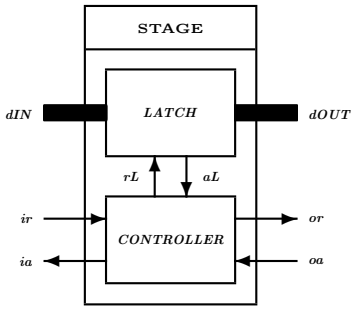


Figure 10: Latch and Controller interplay

The basic pipeline building block is the *stage* which is the composition of a *latch* and its *controller* as shown in Figure 10. The *latch* has two safe states: *open* in which it will admit the fresh data value pending on *dIN* and *closed* when it holds its current value steady both internally and on *dOUT*. The *controller* is responsible for the safety of the open and close operations. Signal lines *ir*/*ia* enable *Input request/acknowledge* communication with a source on the left; signal lines *or*/*oa* enable *Output request/acknowledge* communication with a successor on the right. Linear pipelines are built by abutment with or_k wired to ir_{k+1}, $dOUT_k$ to dIN_{k+1}, and ia_{k+1} to oa_k (see Figure 11).

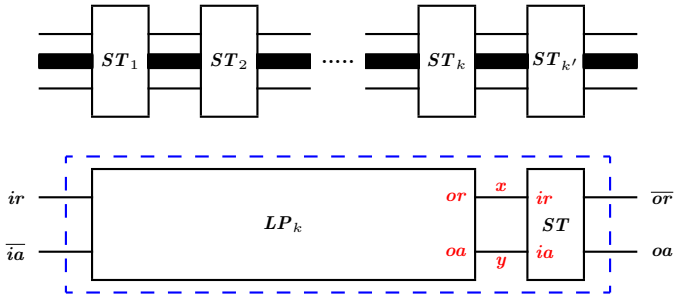


Figure 11: Building linear pipelines by abutment

The construction of such linear pipelines has the straightforward recursive definition:

$$\begin{aligned} LP_1(ir,ia,or,oa) &= ST(ir,ia,or,oa) \\ LP_{k+1}(ir,ia,or,oa) &= (LP_k(ir,ia,x,y) \mid ST(x,y,or,oa)) \setminus \{ x , y \} \end{aligned}$$

with the request and acknowledge lines between LP_k and LP_{k+1} being renamed x and y respectively and hidden (syntactically $\setminus \{ x , y \}$) in the definition of LP_{k+1} so that no other circuit can access them.

3.1 Components for Building a Furber Stage

In this subsection we give signal level specifications of a latch and the basic circuits required to build a specific 2phase pipeline stage presented in Furber [20].

3.1.1 The Latch Model

The latch model in Figure 12 has an input bus *dIN* and output *dOUT* and two control lines *rL* and *aL* which allow it to be *opened* and *closed*.

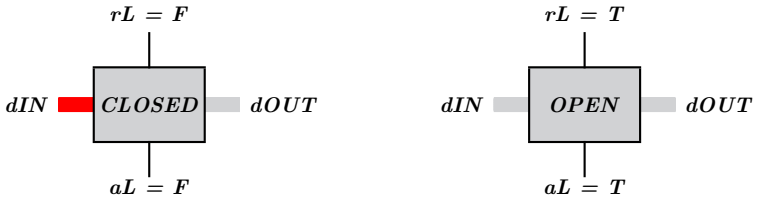


Figure 12: Open and closed latch states

There are two major latch disciplines: *NC* normally closed and *NO* normally open. In this account, we follow the NC discipline only. Furber [20, section 3, page 221 *et. seq.*] provides clear descriptions of both protocols, their uses and their pitfalls.

When an *NC* latch is in its quiescent closed state, both *rL* and *aL* are low and bus *dIN* is disconnected from the latch. We assume that the next data value to be captured, say *Dk*, has been placed on the input bus *dIN*.

open: when *rL* is raised, *dIN* is connected to the latch. The data value on *dIN*, *Dk*, enters the latch and is also copied onto *dOUT*. When these are *both* stable, the latch sends an acknowledgement to the controller by raising *aL*.

When the latch is open, $rL = aL =$ T and $dIN = latch = dOUT$. Whilst open *any* variation in *dIN* will be passed through changing the value in the *latch* and on *dOUT*.

closed: when *rL* is lowered, *dIN* is disconnected from the latch. The data value is now said to be *captured*. The latch then sends an acknowledgement signal informing the controller that the latch is closed by lowering *aL*.

When the latch is closed, $rL = aL =$ F, $latch = dOUT$ and will not be affected by any change on *dIN*.

Note that data is guaranteed stable in the latch and on *dOUT* only when the latch is closed. Since we map signals to wires, the specification of *NC* (with trace variables) is simply:

$$NC = rL.\mathsf{open}.\overline{aL}.rL.\mathsf{closed}.\overline{aL}.NC$$

State machines are quite smart at figuring out whether a signal is to go up or down.

3.1.2 Common 2phase Building Blocks

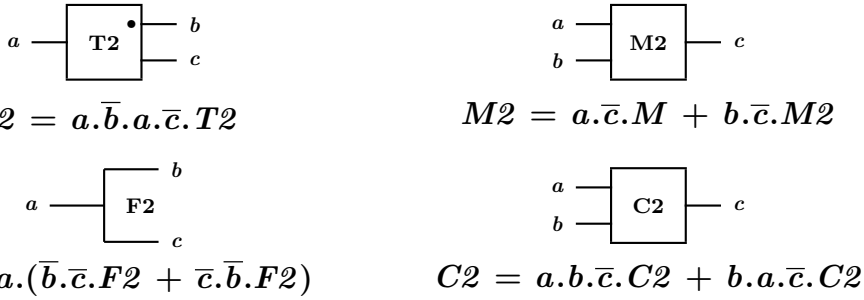


Figure 13: T2/M2 and F2/C2 circuit descriptions

Toggle: $T2 = a.\overline{b}.a.\overline{c}.T2$ The *Toggle* inputs a signal on a and routes it out on b. On receiving a second signal on a, it routes it out on c. Then this 4-cycle repeats. Thus odd numbered input signals are routed out via b and even numbered input signals via c. It is customary to mark the first output line b of *Toggle* with a blob • in a circuit diagram.

Merge: $M2 = a.\overline{c}.M2 + b.\overline{c}.M2$ If a *Merge* receives its next input on a it will acknowledge by emitting a signal on c. If it receives its next input on b it will also acknowledge by emitting a signal on c. The designer must see to it that the $a.\overline{c}$ and $b.\overline{c}$ transactions never overlap.

Fork: $F2 = a.(\overline{b}.\overline{c}.F2 + \overline{c}.\overline{b}.F2)$ The $F2$ has one input a and two outputs b and c. A signal arriving on a is routed to both b and c in turn but in either order. The $F2$ is used to instigate 2 components running in parallel by firing up both.

Collector: $C2 = a.b.\overline{c}.C2 + b.a.\overline{c}.C2$ The $C2$, commonly referred to as a C-element, has two inputs a and b and one output c. When signals have arrived on both its inputs, again in either order, it forwards an acknowledgement on c. The $C2$ element is used as a collector to ensure that the components in a 2-parallel system rendezvous. In general, input signals may be retracted, but as this will not occur in the implementation described, we use a simpler (sufficient) description of $C2$.

3.1.3 FNC: Furber's Normally Closed stage

FNC is constructed by wiring together a C2, an M2, a normally closed latch, a T2, and an F2 fork as shown in Figure 14. For space reasons, the design is presented with the Input lines at the top and the Output lines on the bottom. The C2 ensures that the previous data value has

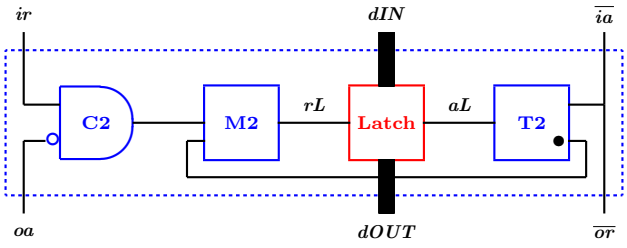


Figure 14: FNC: Furber's normally closed latch

been passed on $dOUT$ **and** that the next data value is stable on dIN. Thereafter a linear thread of control passes through the inner circuitry M2..Latch..T2 twice as shown below in Figure 15.

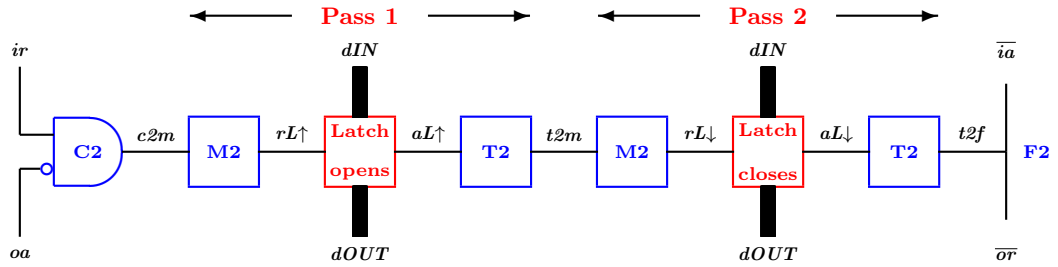


Figure 15: The route through FNC

Figure 15 emphasises that C2 ensures both inputs have fired before the circuit starts to deal with the next request and only on request completion does the F2 distribute the appropriate output signals. The first pass opens the Latch enabling the fresh data value on dIN to be captured. The second pass closes the latch capturing the fresh data value and holding it stable until it has been passed to the successor stage. After the second pass the thread passes to the fork F2 which forwards notice of fresh data to the successor stage and successful capture to the preceding stage.

1. **C2** has input signals on (i) ir and (ii) oa. While ir signals the start of transaction $k+1$, oa signals the completion of transaction k, so that on first use, we need to pre-fire the oa input. This is signified by the circle on its input port in Figures 14 and 15. *Both* being accepted will guarantee that (i) fresh data is present on dIN *and* (ii) that the previous data (still held in the latch and on $dOUT$) has been passed to and captured by the successor stage. Once both signals have been accepted, C2 signals M2.

2. **Pass 1: open the latch**

   (a) **M2** forwards the signal from C2 to the Latch by raising rL.
   (b) **Latch** When $rL\uparrow$ arrives, the Latch opens connecting to dIN and accepts the fresh value from dIN and copies it onto $dOUT$. When both are stable, the *Latch* forwards an acknowledgement to T2 by raising aL.
   (c) **T2** routes this signal back to the **M2** which sets up the second pass to close the latch.

3. **Pass 2: close the latch**

   (a) **M2** forward the request from T2 to the Latch by lowering rL. (Note that the circuitry guarantees that the two uses of M2 are well separated in time and so it is safe to use a Merge here.)
   (b) **Latch** When $rL\downarrow$ arrives, the Latch closes, disconnecting dIN and capturing its fresh data value. The preceding stage may now put a fresh value on dIN without affecting either the local latch value or $dOUT$. The Latch now lowers aL in acknowledgement of the capture.
   (c) **T2** routes the signal to **F2** signalling that the latch is closed guaranteeing the stability of fresh data both locally and on $dOUT$.

4. **F2** sends signals (i) backwards to the inputting stage acknowledging capture of this data value and permitting the next input cycle to begin and (ii) to the successor stage informing it that its next data value is stable on its input bus.

3.2 Specifying max_1

Figure 16 shows the two constituents of Furber's pipeline stage: a normally closed latch and its associated controller.

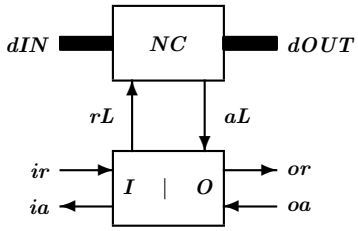


Figure 16: Latch and Controller interplay

Controller Initially we ignore trace variables and any interactions with the Latch. I defines the cyclic sequence of actions on the incoming side and O describes the cyclic sequence of actions on the outgoing side. Their separate descriptions follow the patterns established in OI in Section 2.3 but they cooperate 'in reverse order'. The specification $I = ir\ .\ \overline{ia}\ .\ I$ is to be interpreted as:

***ir*:** when ready, the source places fresh data Dk on dIN and then signals its presence by signalling on line ir. I is then **blocked** until certain that the previous datum $Dk\text{-}1$ has been passed to the next stage.
$\overline{ia}$**:** only when Dk has been captured may I return an acknowledgement to the source by signalling on line ia.
***I*:** whereupon the next input cycle begins

O, which acts as source for the next stage, gives the cycle of actions on the outgoing side: $O = \overline{or}\ .\ oa\ .\ O$. To be safe, we will have to guarantee that O is **blocked** from emitting $\overline{or}$ until a fresh data value has been captured by I. Our initial specification of a stage (without blocking) is just the composition of its input interface with its output interface:

$$\begin{array}{lcl} I & = & ir\ .\ \overline{ia}\ .\ I \\ O & = & \overline{or}\ .\ oa\ .\ O \\ ST & = & (\ I\ |\ O\ ) \end{array}$$

Reading **.** as *and some time later*, we interpret their occurrences in $ir.\overline{ia}$ and $oa.\overline{or}$ as arbitrary *internal* delays; and in $\overline{ia}.ir$ and $\overline{or}.oa$ as arbitrary *external* delays.

The blocking constraints between I and O are handled by two tokens which one may *put* or *get*. We use tokens $NEXT$ with $\overline{gN}/\overline{pN}$ and $PASS$ with $\overline{gP}/\overline{pP}$ operations[1]. The putter is not delayed; the getter may be. The essential safety blockings are:

$$\begin{array}{rcl} NEXT & = & \overline{gN}.\overline{pN}.NEXT \\ PASS & = & \overline{pP}.\overline{gP}.PASS \\ I & = & ir\ .\ gP\ .\ \overline{ia}\ .\ I \\ O & = & gN\ .\ \overline{or}\ .\ oa\ .\ O \\ ST & = & (\ I\ |\ NEXT\ |\ PASS\ |\ O\ ) \end{array}$$

where gP will block I from capturing the next value until the current value has been passed thus preventing premature overwriting, and gN will block O from forwarding its $\overline{or}$ until a fresh

[1] I and O do not handshake directly but indirectly via $NEXT$ and $PASS$. It saves on clutter if we put all the over-barred handshake signals within the two tokens.

value has been captured. All that remains is to free I by a judiciously placed pP and to free O by a judiciously placed pN. The key to these synchronisations is:

$$\begin{array}{rcl} I & = & ir \;.\; gP \;.\; \mathsf{capt} \;.\; pN \;.\; \overline{ia} \;.\; ir \;. \qquad gP \;.\; \mathsf{capt} \;.\; pN \;.\; \overline{ia} \;.\; ir \;. \qquad gP \;.\; \ldots.\; I \\ & & \Downarrow \qquad\qquad\qquad\qquad \Uparrow \qquad\quad \Downarrow \qquad\qquad\qquad\quad \Uparrow \\ O & = & \qquad\quad gN \;.\; \overline{or} \;.\; oa \;.\; \mathsf{pass} \;.\; pP \;. \qquad\quad gN \;.\; \overline{or} \;.\; oa \;.\; \mathsf{pass} \;.\; pP \;.\; \ldots.\; O \end{array}$$

in which we have *temporarily* included trace variables capt and pass for extra clarity. If we omit the handshaking signals and retain only the traces and the get/put operations, we see that these synchronisations faithfully uphold safety by maintaining the cyclic ordering:

$$\textbf{cycle}\ (\ gP \prec \mathsf{capt} \prec pN \prec gN \prec \mathsf{pass} \prec pP\ )$$

(Controller | Latch interplay) Our final step is to include the interactions with the latch. Taking a cue from the thread in Section 3.1.3, we choose to associate the latch open and close requests with our thread I.

$$\begin{array}{rcl} NC & = & \qquad\quad rL \;.\; \mathsf{open} \;.\; \overline{aL} \;.\; rL \;.\; \mathsf{closed} \;.\; \overline{aL} \;.\; NC \\ & & \qquad\quad \Updownarrow \qquad\quad \Updownarrow \quad \Updownarrow \qquad\quad \Updownarrow \\ I & = & ir \;.\; gP \;.\; \overline{rL} \;. \qquad . \; aL \;.\; \overline{rL} \;. \qquad . \; aL \;.\; pN \;.\; \overline{ia} \;.\; I \\ O & = & gN \;.\; \overline{or} \;.\; oa \;.\; pP \;.\; O \\ FNC & = & (\ NC \mid I \mid NEXT \mid PASS \mid O\ ) \setminus \{\ rL,\ aL,\ gN,\ pN,\ gP,\ pP\ \} \end{array}$$

Table 5: First specification of *FNC*

This is observationally equivalent to:

$$\begin{array}{rcl} I & = & ir \;.\; gP \;.\; \mathsf{open} \;.\; \mathsf{closed} \;.\; pN \;.\; \overline{ia} \;.\; I \\ O & = & gN \;.\; \overline{or} \;.\; oa \;.\; pP \;.\; O \\ FNC & = & (\ I \mid NEXT \mid PASS \mid O\ ) \setminus \{\ gN,\ pN,\ gP,\ pP\ \} \end{array}$$

Dropping the trace variables which have served their purpose, we arrive at our final form of the specification of an FNC stage:

$$\begin{array}{rcl} I & = & ir \;.\; gP \;.\; pN \;.\; \overline{ia} \;.\; I \\ O & = & gN \;.\; \overline{or} \;.\; oa \;.\; pP \;.\; O \\ FNC & = & (\ I \mid NEXT \mid PASS \mid O\ ) \setminus \{\ gN,\ pN,\ gP,\ pP\ \} \end{array}$$

Table 6: Final specification of *FNC*

We claim that this specification is the least constrained since I is freed up (by pP/gP) as soon as O has received the acknowledgement oa confirming the current value has been successfully passed downstream, and O is freed up (by pN/gN) as soon as I has captured the next fresh data value. From now on, we will name it max_1.

Rider. Because open/close signals are entirely internal handshakes, the controller and the stage are observationally equivalent. For the same reason, normally open and normally closed variations on the same controller have the same shape. See for example the 4 variations given by Efthymiou and Garside in [19].

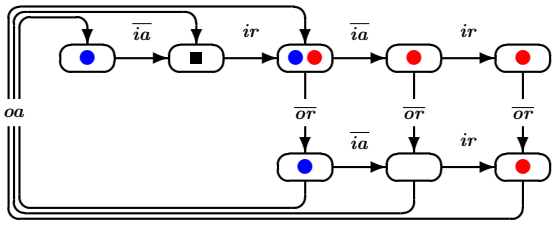


Figure 17: 8-state 2phase latch controller states

3.2.1 Minimised FSM for max_1

The CWB will minimise our formal definition of max_1 into the smallest equivalent state machine as in Figure 17. Horizontally the labels show input signals; vertically and wrapped around, we have the output signals. The initial state is marked ■. The leftmost state in row 1 shows underrun when the output side gets ahead of the input side. The states at the right hand ends of both rows show overrun when the input side is getting ahead of the output side. max_1 permits two extreme behaviours and any interleaving behaviour between them:

1. The south and west edges (output is the faster) has the trace: **LOOP**$(ir.\overline{or}.oa.\overline{ia})$ and may cycle forever through these 4 actions starting from the initial state.
2. The north and east edges (the input side is faster) has the trace: ir.**LOOP**$(\overline{ia}.ir.\overline{or}.oa)$ which takes a lead move to enter the 3rd state in row 1 of max_1 from whence it may cycle forever through the 4 loop actions.

Both loops contain 4 actions (two inputs and two outputs) and maintain the integrity of their signal orderings.

3.2.2 Useful Liveness Properties for max_1

L1 : It is always possible to return to the initial state ■

L2 : In a live shape, the input and output interface cycles are always maintained. Formally, 2cycle (ir, $\overline{ia}$) **and** 2cycle ($\overline{or}$, oa)

Liveness properties **L1** *and* **L2** *are maintained by all live cutaway subdesigns.*

3.3 Experiments with max_1

Experiments were carried out composing shapes max_1 to form linear pipes LP_d (Figure 18) with depths d varying from 1..12 and with parallel pipes PPw,d with widths w varying from 1..8 and depths d from 1..12. We may compare and relate all our designs since pipeline stages, linear pipelines and parallel pipelines, all use the same input/output signal names.

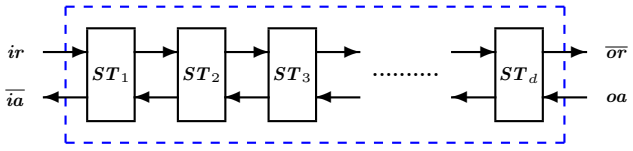


Figure 18: *LPd*: linear pipeline of depth *d* stages

3.3.1 MAX_d: linear pipelines composed from max_1.

2phase pipelines constructed from max_1 stages grow in regular fashion with state sizes 8, 12, 16, ... for pipes of length 1, 2, 3, ...

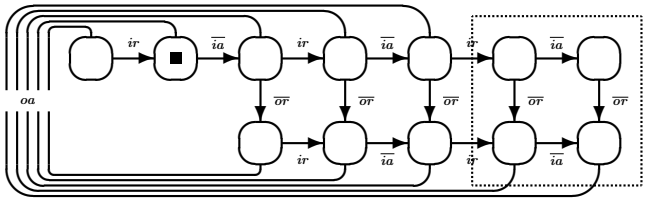


Figure 19: MAX_2: Linear pipeline built from two max_1 stages

Figure 19 shows the minimised state machine for depth 2 with the 4 extra states over max_1 inside a dashed box. As d increases, the shape (profile) of LP_d remains the same: it increases each time by 2×2 states. This indicates *full capacity*: each stage added permits an extra data value to be stored.

Once the pattern of arrows in Figure 19 has been absorbed, a less cluttered picture of max_1 suffices which represents the initial state by ■ and the other live states by •. This clutter free notation is easy to extend to arbitrary pipelines:

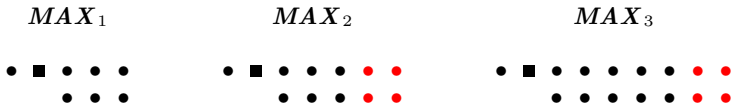


Table 7: Clutter free depiction of MAX pipelines

With a little artistic license, we may indicate the shape of the pipeline MAX_d built from d copies of shape max_1 by

$$\begin{array}{lcll} MAX_1 & = & max_1 & \\ MAX_{d+1} & = & MAX_d & ++ \end{array}$$

Table 8: Pipelines composed with max_1

3.3.2 $PP_{w,d}$: parallel pipelines composed from MAX_d

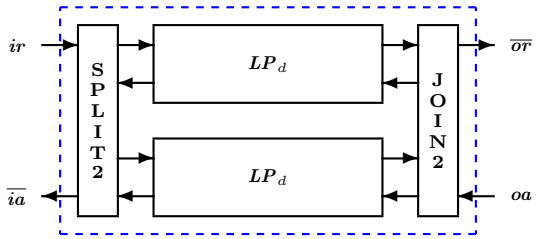


Figure 20: $PP_{2,d}$: parallel pipeline of width 2 and depth d

The parallel pipeline shown in Figure 20 is composed from two inner linear pipes of the same depth d. Entry and exit from the linear pipes is controlled by a *SPLIT2* and a *JOIN2*

respectively. The *SPLIT2* ensures that on receiving a request on *ir*, appropriate data is fed to both inner pipes each of which will acknowledge when their own individual datum has been captured. When both inner acknowledgements have been received, the *SPLIT2* will acknowledge on *ia*. The *JOIN2* works in analogous fashion. Thus each inner pipe will hold the same number of data values but they progress independently. *SPLIT2* and *JOIN2* are implemented as combinations of the *F2* and *C2* circuits defined in Section 3.1.2. Note that if we run parallel pipeline experiments with inner linear pipelines of different depths, then the shortest depth dominates.

We ran our experiments using inner linear pipelines composed from max_1, that is when $LP_d = MAX_d$. For $w = 1..8$ and $d = 1..12$, $PP_{w,d}$ is observationally equivalent to $PP_{1,d}$ which again is observationally equivalent to MAX_d itself. This result has immediate practical application in designing microprocessors where we may replace a complicated data path composed from max_1 stages by a much simpler equivalent model when reasoning about and verifying the rest of the design.

4 Cuts and the Design Space

Taking our cue from the first liveness property **L1** (the initial state must be retained), cutaways can be partitioned into two sets: $\mathcal{R}$ for the input cuts and $\mathcal{L}$ for the output cuts. In this section we show how a complete family of sub-shapes can be generated from max_1 by systematically cutting away input and output states and display both cut lattices. Shapes may be characterised by their cuts from MAX_1 and homogeneous pipelines by their cuts from MAX_d. Experiments show that pipeline patterns are regular and predictable.

4.1 $\mathcal{L}$: Output cuts

In Figure 21 we have replicated the top row of 5 states at the bottom as an extra third row aligned two states to the right. The potential candidates for a left cut now lie in the two states marked ●. If we try to take away more states on the left, then we lose the ability to return to the initial state.

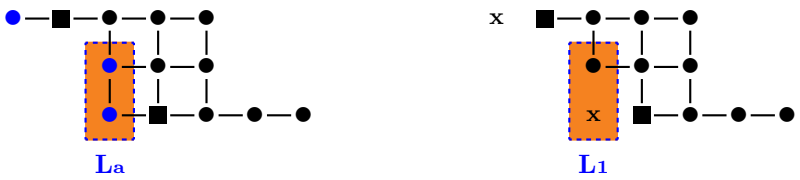


Figure 21: Region of $\mathcal{L}$ cuts from max_1 together with specific cut $\mathbf{L_1}$

Cut $\mathbf{L_a}$ denotes the removal from max_1 of a states from column a working vertically from the bottom (as shown). The specific cut $\mathbf{L_1}$ is depicted on the right.

$\mathbf{L_a}$ constraints: $a \in \{0..2\}$; $\mathbf{L_0..L_2}$

The three L.cuts are displayed below shape by shape. In this figure, ■ represents the initial state, cut states by **x** and uncut states in the L.cut region by ●.

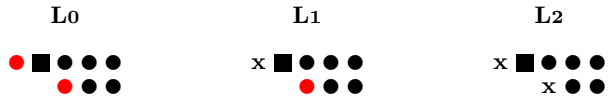


Figure 22: Set of L.cuts

Each L.cut has an distinct identifying signature whose flavour is indicated below. It is straightforward to map our informal signatures into formal modal-μ, the property checking language supported on the CWB.

L0	=	**LOOP**	(	ir	.		$\overline{or}$	.		oa	.	$\overline{ia}$)
L1	=	**LOOP**	(	ir	.		$\overline{or}$	.	**ONLY** $\overline{ia}$		.	oa)
L2	=	**LOOP**	(	ir	.	**ONLY** $\overline{ia}$		.		$\overline{or}$	.	oa)

Table 9: Characteristic L.cut patterns

Starting from the initial state, we track the 4 actions which enable us to loop back (and repeat forever). **ONLY** is used when we deviate from the pattern for **L**0. An output move has been cutaway and our only move is sideways along an input arc. Notice that each loop body contains 2 input actions and 2 output actions and they preserve the mandatory liveness ordering.

4.2 $\mathcal{R}$: Input cuts

Cut $\mathbf{R_{yz}}$ denotes the removal from max_1 of y states from row 1 and z states from row 2 working horizontally from right to left. The maximal right cutaway per row is 2: if we cutaway more we will generate a deadlocked shape. Cut $\mathbf{R}_{21}$ is depicted in Figure 23.

Figure 23: Region of $\mathcal{R}$ cuts from max_1 together with specific cut $\mathbf{R}_{21}$

$\mathbf{R}_{yz}$ cannot choose y and z independently. For example, $\mathbf{R}_{01}$ would render the rightmost state in row y unreachable. The family of all valid $\mathcal{R}$ cuts from max_1 is generated by:

$\mathbf{R}_{yz}$ constraints: $y,z \in \{0..2\}$; $y \geq z$ $\mathbf{R}_{00}..\mathbf{R}_{22}$

The six R.cuts are displayed below shape by shape. In this figure, ■ represents the initial state, cut states by **x** and uncut states in the L.cut region by ●.

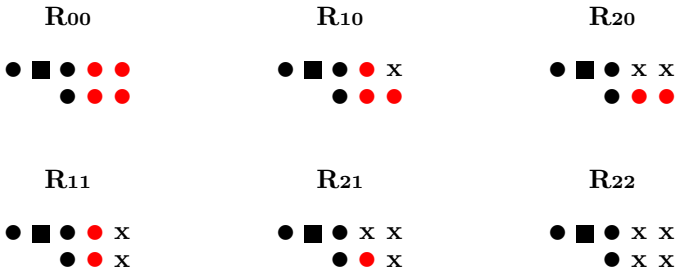


Figure 24: Set of R.cuts

Each R.cut has a distinct signature:

R00	=	*ir*	.	**LOOP**	(		$\overline{ia}$	.		*ir*	.	**ONLY**	$\overline{or}$	.	**ONLY**	*oa*	)
R10	=	*ir*	.	**LOOP**	(		$\overline{ia}$	.	**ONLY**	$\overline{or}$	.		*ir*	.	**ONLY**	*oa*	)
R20	=	*ir*	.	**LOOP**	(	**ONLY**	$\overline{or}$	.		$\overline{ia}$	.		*ir*	.	**ONLY**	*oa*	)
R11	=			**LOOP**	(		*ir*	.		$\overline{ia}$	.	**ONLY**	$\overline{rr}$	.	**ONLY**	*oa*	)
R21	=			**LOOP**	(		*ir*	.	**ONLY**	$\overline{or}$	.		$\overline{ia}$	.	**ONLY**	*oa*	)
R22	=			**LOOP**	(		*ir*	.	**ONLY**	$\overline{or}$	.	**ONLY**	*oa*	.		$\overline{ia}$	)

Table 10: Characteristic R.cut patterns

Starting from the initial state, we track the actions which enable us to loop forever. This time **ONLY** is used when an input move has been cutaway and our only move is along an output arc. Notice that R.cuts $\mathbf{R}_{y0}$ do not loop around the initial state but, after a lead-in *ir* move, from its right neighbour. Again each loop body contains 2 input actions and 2 output actions. Together with the lead in move, if any, they preserve the mandatory liveness ordering.

4.3 Cut lattices

The cuts can be arranged as regular lattices as shown below.

Figure 25: Lattice structures

4.4 Representing a shape

We denote the shape arising from the combination of cuts $\mathbf{L}_a$ and $\mathbf{R}_{yz}$ over max_1 by $\mathbf{L}_a[1]\mathbf{R}_{yz}$. Clearly $max_1 = \mathbf{L}_0[1]\mathbf{R}_{00}$. The core shapes in this design space are shown in Table 11, of which 14 are live and 4 are dead.

Table 11: Complete $\mathcal{L}\times\mathcal{R}$ Cut Table

	Full				Half	Const
La[1]Ryz	**R00**	**R10**	**R11**	**R20**	**R21**	**R22**
L0	live	live	live	live	live	live
L1	live	live	live	live	live	dead
L2	live	live	live	dead	dead	dead

4.5 Homogeneous Pipeline Experiments

In this subsection we show how homogeneous pipelines grow across the whole $\mathcal{L}\times\mathcal{R}$ (3×6) design space. We carried out the same linear and parallel homogeneous pipeline experiments for all 18 core shapes as we did for max_1 in Section 3.3. The experimental results over this range are summarised below:

1. Live shapes compose into live pipelines; dead shapes always compose to dead pipelines.
2. The design space is well behaved. Its 14 live shapes have occupancy full, half, or constant only. Full occupancy means that the pipeline can hold an extra data item for each stage added; half occupancy per 2 stages added; and constant that however long the pipeline, it will only hold one data item. Occupancy is determined solely by the right cut: $\mathbf{R}_{00}$, $\mathbf{R}_{10}$, $\mathbf{R}_{11}$, $\mathbf{R}_{20}$ always give full occupancy; $\mathbf{R}_{21}$ half; and $\mathbf{R}_{22}$ constant.
3. The homogeneous cut set is closed. If we generate pipes from core shapes then the resulting pipeline is cut from MAX_d. We may thus extend our shape notation to pipelines of any depth such that

   SHAPE uniquely as $\mathbf{L}_a[1]\mathbf{R}_{yz}$ PIPE as $\mathbf{L}_a[d]\mathbf{R}_{yz}$

   where $\mathbf{L}_a \in \mathcal{L}$ and $\mathbf{R}_{yz} \in \mathcal{R}$ and d = 1,2,3,.... The uniqueness snag for pipes lies with cuts $\mathbf{R}_{00}$ and $\mathbf{R}_{22}$ since $\mathbf{L}_a[d+1]\mathbf{R}_{22}$ is observationally equivalent to $\mathbf{L}_a[d]\mathbf{R}_{00}$. Here we content ourselves just with the observation. The cure is found in the next section when we generalise to mixed pipelines.
4. Both L.cuts and R.cuts show persistent behaviours (informally retain shape) as pipeline depths increase. This is exemplified by pipelines composed from full, half and constant capacity shapes in Figure 26.

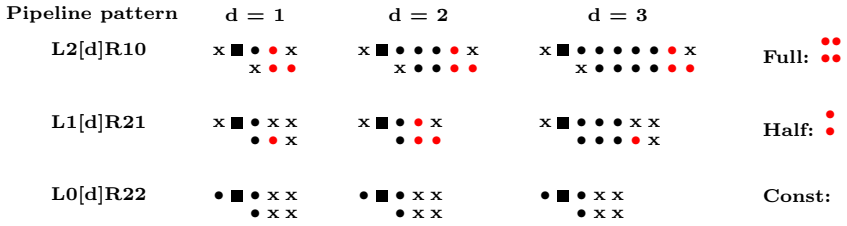


Figure 26: Persistent behaviour of cuts

5. As with max_1, for any live shape a parallel pipeline $PP_{w,d}$ always behaves as its $PP_{1,d}$. But all *full capacity* homogeneous parallel pipelines of depth d behave as $\mathbf{L}_0[d]\mathbf{R}_{00}$; all *half capacity* homogeneous pipelines stutter: those of depth 2d-1 and 2d both behave as $\mathbf{L}_0[d]\mathbf{R}_{00}$; and *all constant* homogeneous pipelines of depth d parallel behave as the shape $\mathbf{L}_0[1]\mathbf{R}_{22}$. In other words, there are only three possible parallel behaviours for homogeneous parallel pipelines composed from the 14 live basic shapes and each of them has a left cut of $\mathbf{L}_0$.

4.5.1 Characterising homogeneous pipelines

It is easy to extend the notation for shape signatures to pipes. The left signature $\mathcal{L}$ remains as it was. The right signature $\mathcal{R}$ has to account for 4 extra states for a full occupancy cut, 2 for a half cut; and remain as is for constant capacity. We follow the style for R.cuts $\mathbf{R}_{00}$, $\mathbf{R}_{10}$, $\mathbf{R}_{20}$ in Table 10, and we prefix the definition at the previous depth by $ir.\overline{ia}$ for full capacity; alternate $\mathbf{R}_{21}$ and $\mathbf{R}_{01}$ cuts at the same depth for half cuts; and use the same check when constant.

LPkLa	=	**La**						
LP1R10	=	**R10**	**LP1R21**	=	**R21**	**LP1R22**	=	**R22**
LP2R10	=	$ir.\overline{ia}$.**LP1R10**	**LP2R21**	=	**R10**	**LP2R22**	=	**R22**
LP3R10	=	$ir.\overline{ia}$.**LP2R10**	**LP3R21**	=	$ir.\overline{ia}$.**LP1R21**	**LP3R22**	=	**R22**
LP4R10	=	$ir.\overline{ia}$.**LP3R10**	**LP4R21**	=	$ir.\overline{ia}$.**LP2R10**	**LP4R22**	=	**R22**
.....			.....			.....		

Table 12: Sample linear pipeline characterisations

5 Mixed Pipeline Structures and Patterns

In this section we relate experiments carried out with linear mixed pipelines. The increase in generality yielded one surprise: some R.cut combinations, for example $\mathbf{R}_{11}.\mathbf{R}_{20}$, gave rise to a right cut of $\mathbf{R}_{31}$ (valid from depth 2). With $\mathbf{R}_{31}$ included however, the R.cuts become closed. This slight addition has important ramifications. By dispensing with cut $\mathbf{R}_{22}$, the uniqueness of pipeline representation snag noted in the previous section disappears. Further the lattice of R.cuts may be cast as two related chains each of which has a simple mapping from the chain of L.cuts; a structure that we exploit in this section.

Notation. In this section, we use $Sa.Sb$ as a compact notation for the linear pipeline formed by Sa and Sb. In the same way, if Sa has cuts $\mathbf{L}_a$ and $\mathbf{R}_a$ and Sb has cuts $\mathbf{L}_b$ and $\mathbf{R}_b$, then we denote the cuts of $Sa.Sb$ by $\mathbf{L}_a.\mathbf{L}_b$ and $\mathbf{R}_a.\mathbf{R}_b$.

5.1 Initial mixed experiments

For linear pipeline $S2$ constructed from shapes Sa and Sb (Figure 27), signal $\overline{or}$ from Sa handshakes with ir of Sb and signal $\overline{ia}$ from Sb handshakes with oa of Sa. The liveness conditions thus assure the liveness of the pipeline provided that Sa and Sb are both live, be they of full, half or constant capacity.

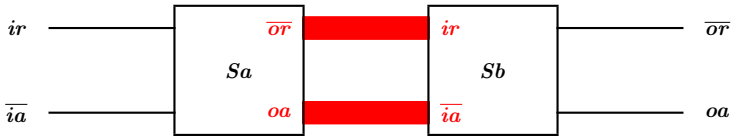


Figure 27: Mixed pipeline $S2 = Sa.Sb$

Revised R.cuts. Initially we experimented with all 18 combinational possibilities for L.cuts and R.cuts in pipelines of depth 2. They confirmed the liveness proposition and the independence of L.cuts and R.cuts, but yielded one unexpected result: that the R.cuts were not closed. The R.cut combinations $\mathbf{R}_{11}.\mathbf{R}_{20}$, $\mathbf{R}_{21}.\mathbf{R}_{20}$, $\mathbf{R}_{11}.\mathbf{R}_{21}$ all combine to form a depth 2 R.cut of $\mathbf{R}_{31}$. If we augment the set of right cuts by $\mathbf{R}_{31}$, then R.cuts becomes closed. We have already noted that $\mathbf{R}_{22}$ is redundant. If we omit it, we have an R.cut set that is closed.

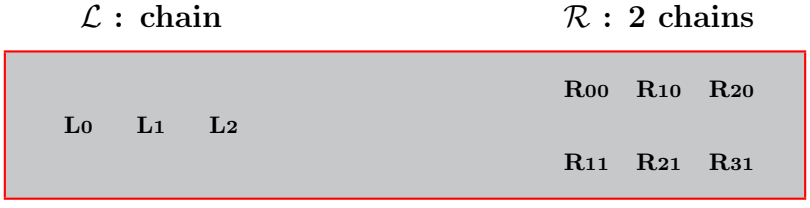


Figure 28: Lattice structures revised

Since the L.cuts form a chain, it is fruitful to think of the R.cuts as a lattice structure formed by two related chains R0 and R1 shown horizontally on the right of Figure 28. It is then easy to map amongst them and take advantage of the structural relationships between the output and input cuts.

Revised pipeline notation. If we extend our definition of maximal pipelines to include the the 4 state max_0 (= $\mathbf{L}_0[1]\mathbf{R}_{22}$)

$$\begin{array}{lcll} \boldsymbol{max}_0 & = & \boldsymbol{ir}.\overline{\boldsymbol{or}}.\boldsymbol{oa}.\overline{\boldsymbol{ia}}.\boldsymbol{max}_0 & \\ \boldsymbol{MAX}_0 & = & \boldsymbol{max}_0 & \\ \boldsymbol{MAX}_{d+1} & = & \boldsymbol{MAX}_d & ++ \end{array}$$

Table 13: MAX pipelines revised

then we can express any live pipeline constructed from core shapes uniquely by cutaways from some MAX_d. Clearly, there is just one live pipeline at depth 0, max_0 itself, which cannot tolerate any cuts. There are 13 live pipelines at depth 1 (where no $\mathbf{R}_{31}$ cut will be live). Pipelines at depths 2 or more accept all 18 cut possibilities.

5.2 Experimental Results for Mixed Linear Pipelines

Our experiments paired up all shape possibilities $\mathcal{L}\times\mathcal{R}$, including the non-live which served to confirm our liveness properties. By enumeration over all cases $S2 = Sa.Sb$:

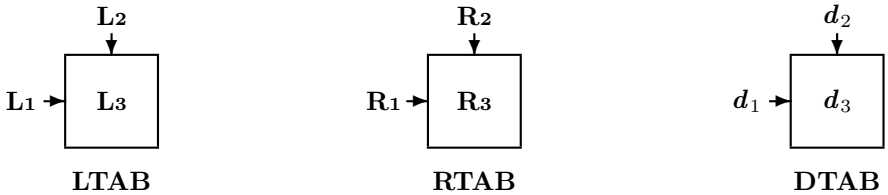
1. **Liveness**: $S2$ is live if and only if Sa and Sb are live.
2. **Independence**: L.cuts and R.cuts are independent.
   $\mathbf{L}_2$, the L.cut of $S2$, depends solely upon $\mathbf{L}_a$ and $\mathbf{L}_b$.
   $\mathbf{R}_2$, the R.cut of $S2$, depends solely upon $\mathbf{R}_a$ and $\mathbf{R}_b$
3. **Closure**: $\mathbf{L}_2 \in \mathcal{L}$ and $\mathbf{R}_2 \in \mathcal{R}$. This implies that any pipeline constructed from our basic shapes can be expressed in terms of cuts from some MAXd for some d.
4. **Unit**: there are unit (identity) $\mathcal{L}$ and $\mathcal{R}$ cuts.

   $$\begin{array}{lcl} \mathbf{L}_1 \,.\, \mathbf{L}_a = \mathbf{L}_a & \qquad & \mathbf{L}_a \,.\, \mathbf{L}_1 = \mathbf{L}_a \\ \mathbf{R}_{10} \,.\, \mathbf{R}_{yz} = \mathbf{R}_{yz} & & \mathbf{R}_{yz} \,.\, \mathbf{R}_{10} = \mathbf{R}_{yz} \end{array}$$

5. **Association**: given the composition of three shapes $Sa.Sb.Sc$

   $$\begin{array}{l} (\,\mathbf{L}_a \,.\, \mathbf{L}_b\,) \,.\, \mathbf{L}_c = \mathbf{L}_a \,.\, (\,\mathbf{L}_b \,.\, \mathbf{L}_c\,) \\ (\,\mathbf{R}_a \,.\, \mathbf{R}_b\,) \,.\, \mathbf{R}_c = \mathbf{R}_a \,.\, (\,\mathbf{R}_b \,.\, \mathbf{R}_c\,) \end{array}$$

Thus the behaviours of L.cuts and R.cuts are independent, consistent and predictable. That L.cuts and R.cuts have a well defined algebraic structure hints at the prospect of further relationships and insights.



5.2.1 Tabulation of LTAB, RTAB and DTAB

Because L.cuts and R.cuts work independently, we can condense their experimental properties in simple lookup tables. The pipeline depth of the resulting shape depends upon the constituent depths and also upon its two R.cuts.

LTAB. Cut $\mathbf{L}_1$ is the unit (or identity) cut.

Table 14: LTAB

La.Lb	L0	L1	L2
L0	L0	L0	L0
L1	L0	L1	L2
L2	L2	L2	L2

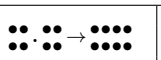
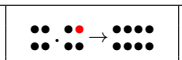
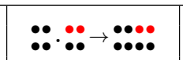
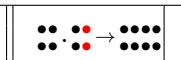
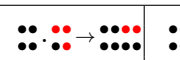
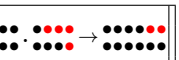
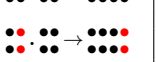
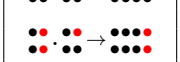
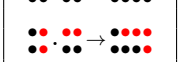
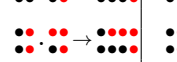
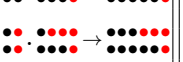
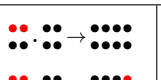
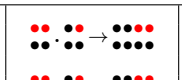
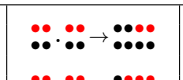
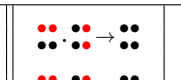
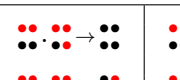
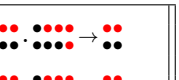
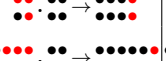
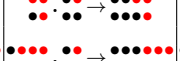
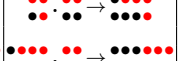
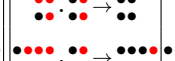
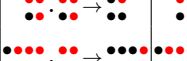
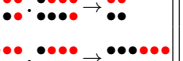
RTAB. Cut $\mathbf{R}_{10}$ is the unit cut. Notice that the row entries for $\mathbf{R}_{21}$ are related to those of $\mathbf{R}_{10}$ and span the spectrum of R.cuts but shifted by 3.

Table 15: RTAB

Ra.Rb	R00	R10	R20	R11	R21	R31
R00	R00	R00	R20	R00	R20	R20
R10	R00	R10	R20	R11	R21	R31
R20	R00	R20	R20	R00	R00	R20
R11	R11	R11	R31	R11	R31	R31
R21	R11	R21	R31	R00	R10	R20
R31	R11	R31	R31	R11	R11	R31

DTAB. The data for DTAB is presented graphically. Since it partly depends upon the constituent R.cuts, R1.R2, the results are entered in patterns of the form R1.R2 → R3.

Table 16: DTAB

Ra.Rb	R00	R10	R20	R11	R21	R31
R00						
R10						
R11						
R20						
R21						
R31						

The NW, NE, SW quadrants contain those combinations that give rise to full occupancy. In these quadrants, if R1 is of depth d1 and R2 is of depth d2 then R1.R2 gives rise to a pipe of length d1+d2. The SE quadrant contains those combinations that give rise to pipes of length d1+d2-1. So DTAB reveals that the calculation of pipeline depth is simple.

5.3 Calculation of Pipeline Behaviours

Space precludes any account of mixed parallel pipeline behaviours, but they too are calculable.

5.3.1 Application I: mixed linear pipe of depth 4

Once we have these tables it is straightforward to calculate the shape of any 2 stage pipeline. Because the cuts are closed, we can calculate the behaviours and properties of pipelines of any depth. Longer pipelines may be calculated by iteration as in the following application:

let	$PIPE4$	=	$(\ S_1 \mid S_2 \mid S_3 \mid S_4\ ) \setminus HIDE$
where	S_1	=	$\mathbf{L_1[1]R_{20}}$
	S_2	=	$\mathbf{L_0[0]R_{00}}$
	S_3	=	$\mathbf{L_2[2]R_{31}}$
	S_4	=	$\mathbf{L_1[1]R_{11}}$
then	$L4$	=	$\mathbf{(L_1.L_0).L_2.L_1 \rightarrow (L_0.L_2).L_1 \rightarrow L_0.L_1 \rightarrow L_0}$
and	$R4$	=	$\mathbf{(R_{20}.R_{00}).R_{31}.R_{11} \rightarrow (R_{00}.R_{31}).R_{11} \rightarrow R_{20}.R_{11} \rightarrow R_{00}}$
and	$d4$	=	$\mathbf{(1.0).2.1 \rightarrow (1.2).1 \rightarrow 3.1 \rightarrow 3}$
Thus	$PIPE2$	=	$\mathbf{L_0[1]R_{00}}$ ***since*** $\mathbf{R_{20}.R_{00}}$ ***lies in the SW quadrant***
	$PIPE3$	=	$\mathbf{L_0[3]R_{20}}$ ***since*** $\mathbf{R_{00}.R_{31}}$ ***lies in the NE quadrant***
	$PIPE4$	=	$\mathbf{L_0[3]R_{00}}$ ***since*** $\mathbf{R_{20}.R_{11}}$ ***lies in the SE quadrant***

Note that associativity allows us to compose these 4 shapes any way we choose as long as their ordering is respected. These predictions have all been confirmed by experiment.

5.3.2 Application II: mixed rings

We can model rings by connecting the outgoing pipe line $\overline{or}$ to ir and the outgoing line $\overline{ia}$ to oa modulo initialising the first ring stage so that it has captured a data value. In our experiments all other pipeline stages are empty.

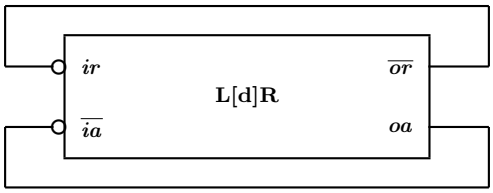


Figure 29: Mixed rings

Experiments show that ring stages may be mixed freely and that the ring will be live provided that the pipeline is live and has depth at least 2.

5.3.3 Application III: relating LTAB and RTAB

In this application we show the relationships between LTAB and each of the 4 quadrants of RTAB. This means we can calculate both L.cut and R.cut pipeline behaviours from LTAB.

The two cut families are related by a number of maps. As we will be operating on cuts, rows of cuts, and (sub-)tables of cuts, we introduce two extra operators. If f is a function operating on a cut, then **ROW**.f applies f to each cut in a row of cuts and **TAB**.f applies f each cut element in a table of cuts. The design space is so small that we define the basic cut functions by enumeration of cases.

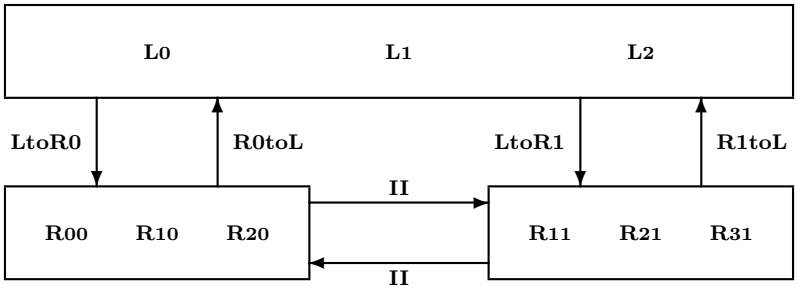


Figure 30: Maps between L.cut and R.cut chains

Maps between R.chain0 and R.chain1: **II** Rxy = R(x+1)(y+1) **mod 22**

ROW.II { $\mathbf{R}_{00}$, $\mathbf{R}_{10}$, $\mathbf{R}_{20}$ } = { $\mathbf{R}_{11}$, $\mathbf{R}_{21}$, $\mathbf{R}_{31}$ }
ROW.II { $\mathbf{R}_{11}$, $\mathbf{R}_{21}$, $\mathbf{R}_{31}$ } = { $\mathbf{R}_{00}$, $\mathbf{R}_{10}$, $\mathbf{R}_{20}$ }

Maps between $\mathcal{L}$ and $\mathcal{R}$:

ROW.LtoR0 { $\mathbf{L}_0$, $\mathbf{L}_1$, $\mathbf{L}_2$ } = { $\mathbf{R}_{00}$, $\mathbf{R}_{10}$, $\mathbf{R}_{20}$ }
ROW.LtoR1 { $\mathbf{L}_0$, $\mathbf{L}_1$, $\mathbf{L}_2$ } = { $\mathbf{R}_{11}$, $\mathbf{R}_{21}$, $\mathbf{R}_{31}$ }

ROW.R0toL { $\mathbf{R}_{00}$, $\mathbf{R}_{10}$, $\mathbf{R}_{20}$ } = { $\mathbf{L}_0$, $\mathbf{L}_1$, $\mathbf{L}_2$ }
ROW.R1toL { $\mathbf{R}_{11}$, $\mathbf{R}_{21}$, $\mathbf{R}_{31}$ } = { $\mathbf{L}_0$, $\mathbf{L}_1$, $\mathbf{L}_2$ }

Complements are symmetry relations between cuts in a family. Notice that there is also symmetry within R.cuts as a whole.

L.COMP	=	{	($\mathbf{L}_0$, $\mathbf{L}_2$)	,	($\mathbf{L}_1$, $\mathbf{L}_1$)	,	($\mathbf{L}_2$, $\mathbf{L}_0$)	}
R0.COMP	=	{	($\mathbf{R}_{00}$, $\mathbf{R}_{20}$)	,	($\mathbf{R}_{10}$, $\mathbf{R}_{10}$)	,	($\mathbf{R}_{20}$, $\mathbf{R}_{00}$)	}
R1.COMP	=	{	($\mathbf{R}_{11}$, $\mathbf{R}_{31}$)	,	($\mathbf{R}_{21}$, $\mathbf{R}_{21}$)	,	($\mathbf{R}_{31}$, $\mathbf{R}_{11}$)	}

Transpose $\text{LTAB}^{\mathbf{T}}$ is required in generating RTAB00, the Ro×Ro quadrant of RTAB. It arises because we take L.cuts vertically and R.cuts horizontally.

FLIP is a subtable operation that takes a 3×3 quadrant, swaps rows 1 and 3 and applies **ROW.II** to row 2. Notice that **II** ∘ FLIP = FLIP ∘ **II**.

FLIP [r1, r2, r3] = [r3, ROW.II r2, r1]

5.3.4 Generation of RTAB from LTAB

We generate RTAB quadrant by quadrant in the chain×chain order: $\mathbf{R}_0$×$\mathbf{R}_0$, $\mathbf{R}_0$×$\mathbf{R}_1$, $\mathbf{R}_1$×$\mathbf{R}_0$, $\mathbf{R}_1$×$\mathbf{R}_1$.

In tabular form

RTAB00	=	**TAB.LtoR0 LTAB$^{\mathbf{T}}$**
RTAB01	=	**FLIP RTAB00**
RTAB10	=	**TAB.II RTAB00**
RTAB11	=	**TAB.II RTAB01**

Table 17: Table of LTAB → RTAB transformations

and by diagram:

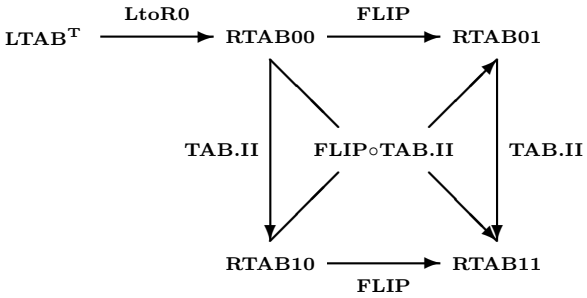


Figure 31: LTAB → RTAB transformation

Notice that the bottom half of RTAB may be obtained from the top half by applying TAB.II and *vice versa*. Going laterally is more interesting.

5.4 Extension to 4phase untimed

The approach to mixed pipelines outlined in this section has been applied equally successfully to the 10×20 untimed subset of 4phase designs; the 19×38 untimed ++ locally timed subset; and to the complete 35×140 4phase design space.

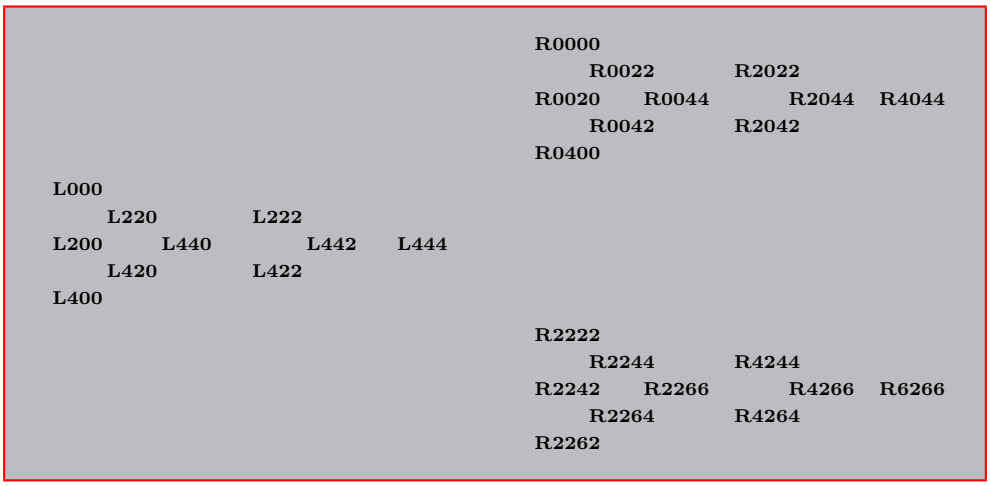


Figure 32: L.wedge R0.wedge and R2.wedge

Just as with 2phase, the R.cuts previously published [8, 34] were closed for homogeneous pipelines but not for mixed. Following through the techniques presented here resulted in a

reduction of in the size of R.cuts from 25 to 20 which revealed for the first time a splitting of R.cuts into two subsets each of which is isomorphic to L.cuts as shown in Figure 32. It is doubtful we would have made these connections without venturing into mixed pipelines and insights gleaned from this work in the 2phase domain.

In 4phase, the L.cuts have 3 indices, $\mathbf{L}_{abc}$ and R.cuts 4, $\mathbf{R}_{wxyz}$. Each R.cut in wedge R0 has x=0, and each R.cut in wedge R2 has x=2. Similar to the **II** operation for 2phase, cut $\mathbf{R}_{w0yz}$ in wedge R0 is vertically aligned with cut $\mathbf{R}_{(w+2)2(y+2)(z+2)}$ in wedge R2.

It is simple to draw up the 4phase equivalents of the 2phase mapping functions of Section 5.3.3. The structure of the 2phase transformations given in Figure 31 still holds subject to a (simply) modified versions of the **FLIP** operation and **II** operations.

6 From Shape to Silicon

An implementation study was performed similar to that in [34]. Systematic concurrency reduction produced by applying cuts to the most concurrent shape results in the complete design space. This allows all possible specifications to be investigated in order to obtain the best circuit for any specific design goal, be it high performance, low power, small area, latency, etc.

When realizing designs, shapes are partitioned into design styles. Shapes can be categorized into two protocol classes, untimed and timed. Untimed shapes come in two classes: delay insensitive (DI) and speed independent (SI). Likewise timed shapes have two classes: locally timed (LT) and externally timed (ET). Locally timed shapes can be designed to work in nearly all homogeneous or mixed pipelines based on the local delays present inside the circuit implementation of any given shape. However, externally timed shapes are only correct when specific relative delay requirements are enforced on the response time of other controllers. For example, the consumer (downstream controller) connected to an ET shape may be required to respond much faster than the producer (the upstream controller).

This paper extends previous work to include all locally timed shapes. LT shapes for the 2phase family are formed by the $\mathbf{L}_1$ and $\mathbf{R}_{10}$ cuts. Externally timed cuts include all $\mathbf{R}_{y1}$ cuts. Thus to investigate all DI, SI, and LT shapes, the $\mathcal{L}$ and top $\mathcal{R}$ chains from Figure 28 are employed; the lower $\mathcal{R}$ chain is discarded. The 3×3 cuts result in 9 shapes categorized as one DI, two SI, five LT and one deadlocked shape.

An automated flow was developed to generate and optimize asynchronous pipeline controllers from the shape state machines. The designs were synthesized and technology mapped to Artisan's static library for IBM's 65nm 10sf process node using Petrify [15]. The circuits

Frequency in GHz

L0	L1	L2	La[1]Ryz
4.22	4.40	4.00	R00
4.50	5.05	4.27	R10
4.20	3.78	–	R20

Energy per Token in pJ

L0	L1	L2	La[1]Ryz
8.73	11.20	7.15	R00
8.90	7.95	7.75	R10
7.48	5.18	–	R20

Area in μm^2

L0	L1	L2	La[1]Ryz
217	243	204	R00
221	220	206	R10
207	211	–	R20

Figure 33: Performance, power, and area of the circuits

are verified for conformance to the shape, and automatic relative timing (RT) constraints are created [43]. The RT constraints are applied using a custom flow to synthesize and optimize the circuits for power and performance using commercial clocked CAD tools such as Design Compiler and SOC Encounter [40]. The results report parasitic extracted values for the physically placed and routed designs.

The results for performance, power and area are shown in Figure 33. The better designs are highlighted in green, the worse designs in red. The timed L.cut and R.cut are highlighted in yellow. All rows and columns employing timed cuts result in timed shapes (5 of the 8).

The NW corner of the tables contains shapes with the most concurrency and the SE corner the most sequential. In general, the more concurrent shapes should admit a higher performance; the most sequential lower power and smaller area. This general trend applies with some notable exceptions. For example, the larger and more complicated circuits required for higher concurrency may hamper performance. But most significantly, the locally timed cuts produce the fastest circuits. The four highest frequency designs all employ timed cuts. The best timed shape produces a circuit that is 20% faster than the circuit from the best untimed shape. Only a handful of timed circuits have been investigated and published in the literature. This study thereby opens up the possibility of uncovering new design sets with substantial performance improvements over current state of the art.

7 Overview and summary

We have investigated an abstract model of latch controllers and presented new experimental results on its outer and inner structure in terms of $\mathcal{L}$ and $\mathcal{R}$ lattices of cuts from the maximal shape. $\mathcal{L}\times\mathcal{R}$ reveals the whole design space and has been used to guide experiments ranging from investigating linear and parallel pipeline patterns through to investigating the behaviour of families of circuits. The patterns have suggested algorithmic rules for predicting the behaviours of homogeneous and mixed pipelines. Such predictions make it possible to replace complicated irregular parallel datapaths by smooth linear pipeline behaviours—a very useful mental model when designing systems.

Novel design space patterns herein described include: complete lattices for $\mathcal{L}$ and $\mathcal{R}$ cuts; complete design space as $\mathcal{L}\times\mathcal{R}$; notation for each shape and each pipeline as $\mathbf{L}_{\mathrm{a}}[\mathrm{k}]\mathbf{R}_{\mathrm{yz}}$; the consistent and independent growth patterns $\mathcal{L}$ and $\mathcal{R}$ cuts; and not least in the treatment of mixed pipelines. Demonstrations of their practical use were given for predicting pipeline behaviours and generating novel circuits.

1. Survey of published designs (4phase). This kept our work grounded. Our common algebraic (FSM) notation for their shapes (design abstractions) enables them to be ordered and compared. In shape format, all published designs had at least input signal ordering constraints or output signal ordering constraints; and usually both. Taking the intersection of constraints over all surveyed shapes revealed max_1, a maximal shape (most concurrent possible signal orderings), from which all published shapes could be constructed by cutting away states on the Left or on the Right.
2. Taking our cue from the way input and output cuts characterised published designs, we fully generalised the cut possibilities from max_1. The product $\mathcal{L}\times\mathcal{R}$ reveals the complete design space for max_1 and its sub-designs. The cut classes form elegant lattices and enable tight mathematical definitions of design style domains over max_1: e.g. delay insensitive, speed independent, burst mode, relative timing. The definitions are defined simply in terms of sublattices of L and R. A software suite has been developed which will take shape specifications and generate characterized circuits. Since we have the design space, and know how to categorize design styles, we can (and have) examined the 3×6 2phase design space and the 10×25 DI subset of 4phase. Work is in progress studying such characteristics as: area, power consumption, speed, ...

3. We have conducted experiments on the behaviours of homogeneous linear pipelines of depths 1..12 and over homogeneous parallel pipelines of depths 1..12 and widths 1..8 over the Untimed sub-design space. These experiments revealed much persistent structured behaviour: examples (i) if a shape is live, then so will be its pipelines; (ii) **L**.cuts and **R**.cuts work independently of each other; (iii) all parallel compositions are observationally equivalent to some DI single pipeline; (iv) shapes exhibit full, half, or unit occupancy; (v) occupancy is determined solely by a shape's **R**.cut. For homogeneous pipelines, the **L** and **R** sets are closed, so that we can adapt the specification of a shape to define a homogeneous pipeline of any depth
4. We have also experimented with mixed pipelines: by which we permit a linear pipeline to have different shapes throughout its length; and with mixed parallel pipelines where we may run distinct homogeneous (or mixed) linear pipelines in parallel. For tractable design subsets, the results are again structured. In particular liveness, cut independence, the DI behaviour of parallel pipes still hold. But for mixed pipelines, the **R** sets are not closed. However it is a simple matter to include extra **R**.cuts (valid only from pipeline depth 2) to ensure closure. Experiments have been completed over 2phase and 4phase shapes. These show that shapes associate under pipelining and that the behaviour of linear pipelines of any depth can not only be specified in terms of the **L** and extended **R**.cuts but can also be calculated from their constituents. By extension, we can also calculate the behaviour of mixed parallel pipelines from their constituents.
5. The understanding and formalisation of this increase generality resulted in some dramatic (and practical) simplifications. We can now relate the **L** and **R** cut lattices and once we have calculated the LTAB, the mixed **L**.cut behaviour table, we can construct RTAB the mixed **R**.cut table by simple and standard transformations.

Acknowledgements

This work has been supported in part by a grant from Sun Microsystems.

Thanks are due to the asynchronous community who have made great efforts to document and explain their circuits so they are clear to the community at large. This body of work enabled us to model real practical designs rather than experiment with a few idealised ones and kept us grounded. Importantly, the corpus was sufficiently large to guide our research directions.

For their individual help and guidance over the years, we thank Erik Brunvand, Bill Coates, Jordi Cortadella, Al Davis, Jo Ebergen, Steve Furber, Jim Garside, Luciano Lavagno, Andrew Lines, Ying Liu, Faron Moller, Mike Stannett, Georg Struth, Chris Tofts, and Alexander Yakovlev. They are of course in no way to blame for any lack of clarity or errors we may have made.

References

[1] L. Aceto, K.G. Larsen, and A. Ingolfsdottir. An Introduction to Milner's CCS. Course Notes for *Semantics and Verification*. Constantly under revision. The most recent version is available at the URL *http://www.cs.auc.dk/∼luca/SV/Intro21ccs.pdf*, BRICS, Department of Computer Science, Aalborg, Denmark, 2005.

[2] J. M. Anderson, W. S. Coates, A. L. Davis, R. W. Hon, I. N. Robinson, S. V Robison, and K. S. Stevens. The Architecture of FAIM-1. *Computer*, 20(1):55–65, January 1987.

[3] E. R. Berlekamp, J. H. Conway, and R. K. Guy. Winning Ways for your Mathematical Plays. pages 598–601, London, 1982. Academic Press.

[4] G. Birtwistle and Y. Liu. Modelling AMULET1 in CCS. *IEE Colloquium on Design and Test of Asynchronous Systems*, pages 9/1–9/6, 1996.

[5] G. Birtwistle and Y. Liu. Specification and Property Checking of the AMULET1 Address Interface. *Designing Correct Circuits 96*, 1996.

[6] Graham Birtwistle. Control states in asynchronous pipelines. In Alex Yakovlev and Reinder Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 45–55, July 2000.

[7] Graham Birtwistle and Matthew Morley. Case study: specifying and property checking TK, an asynchronous AMULET-like microprocessor. In A. Yakovlev and R. Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations (AINT'2000)*, pages 13–22, TU Delft, July, 2000.

[8] Graham Birtwistle and Kenneth Stevens. The Family of 4-phase Latch Controllers. In *ASYNC 2008, 14th International Symposium on Asynchronous Circuits and Systems*, pages 71–82, Newcastle upon Tyne, UK, 7-11 April, 2008.

[9] I. Blunno, J. Cortadella, A. Kondratyev, L.Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronisation. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits*, pages 149–158. IEEE/ACM, April 2004.

[10] A. E. Brouwer, G. Horvath, I. Molnar-Saska, and C. Czabo. On three-rowed Chomp. In *INTEGER: Electronic Journal of Combinatorial Number Theory*, volume 5, pages 1–11, 2005.

[11] E. Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[12] E. L. Brunvand and R. F. Sproull. Translating Concurrent Programs into Delay-insensitive Circuits. In *IEEE International Conference on Computer-Aided Design*, pages 262–265, Los Alamitos, CA, 1989. IEEE Comput. Soc. Press.

[13] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits From Graph-Theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, September 1987.

[14] J. Cortadella, M. Kishinevsky, S. M. Burns, A. Kondratyev, L. Lavagno, K. S. Stevens, A. Taubin, and A. Yakovlev. Lazy transition systems and asynchronous circuit synthesis with relative timing assumptions. *IEEE Transactions on Computer-Aided Design*, 21(2):109–130, Feb 2002.

[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[16] Paul Day and J. Viv Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[17] J. Ebergen. Parallel Computations and Delay-Insensitive Circuits. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 85–104, Berlin, 1991. Workshops in Computing Series, Springer Verlag.

[18] J. C. Ebergen, J. Segers, and I. Benko. Parallel Program and Asynchronous Circuit Design. In A. Davis and G. Birtwistle, editor, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*, pages 50–103. Springer Verlag, Workshops in Computing Series, 1995.

[19] Aristides Efthymiou and Jim D. Garside. Adaptive pipeline structures for speculation control. In *Ninth International Symposium on Asynchronous Circuits and Systems*, pages 46–55. IEEE, May 2003.

[20] S. Furber. Computing without Clocks: Micropipelining the ARM Processor. In A. Davis and G. Birtwistle, editor, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*, pages 211–262. Springer Verlag, Workshops in Computing Series, 1995.

[21] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver. AMULET2e: An Asynchronous Embeded Controller. *Proceedings of IEEE*, 87(2):243–256, February 1999.

[22] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 11–16. IEEE Computer Society Press, March 1996.

[23] Stephen B. Furber. A small compendium of 4-phase macropipeline latch control circuits. Technical Report v0.3, 17/01/99, University of Manchester, Dept. of Computer Science, 1999.

[24] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[25] Rakefet Kol and Ran Ginosar. A doubly-latched asynchronous pipeline. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 706–711, Oct 1996.

[26] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, Pasadena, CA, 1998.

[27] JianWei Liu. Arithmetic and Control Componenets for an Asynchronous System. PhD Thesis, Department of Computer Science, University of Manchester, 1997.

[28] A. J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1:226–234, 1986.

[29] A. J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, New York, 1990. Addison-Wesley.

[30] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, North Holland, 1990.

[31] Alain J. Martin. The Design of a Delay-Insensitive Microprocessor: An Example of Circuit Synthesis by Program Transformation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification, and Synthesis: Mathematical Aspects, Proceedings of the Mathematical Sciences Institute Workshop, Cornell University, Ithaca, N.Y., July, 1989*, pages 244–259, New York, 1989. Springer-Verlag.

[32] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[33] Faron G. Moller and Perdita Stevens. *The Edinburgh Concurrency Workbench (Version 7)*. University of Edinburgh, October 1992.

[34] Santosh Nagasai, Kenneth Stevens, and Graham Birtwistle. Concurrency Reduction of Untimed Latch Protocols – Theory and Practice. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 26–37, Grenoble, France, May 2010.

[35] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 198–209. IEEE, Apr 2000.

[36] Montek Singh and Steven M. Nowick. MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 9–17, Nov 2001.

[37] Kenneth S. Stevens, Ran Ginosar, and Shai Rotem. Relative Timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(11):129–140, February 2003.

[38] Kenneth S. Stevens, Shane V Robison, and A.L. Davis. The Post Office – Communication Support for Distributed Ensemble Architectures. In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 160 – 166, May 1986.

[39] Kenneth S. Stevens, Shai Rotem, Ran Ginosar, Peter Beerel, Chris J. Myers, Kenneth Y. Yun, Rakefet Kol, Charles Dike, and Marly Roncken. An Asynchronous Instruction Length Decoder. *IEEE Journal of Solid State Circuits*, 36(2):217–228, February 2001.

[40] Kenneth S. Stevens, Yang Xu, and Vikas Vij. Characterization of Asynchronous Templates for Integration into Clocked CAD Flows. In *15th International Symposium on Asynchronous Circuits and Systems*, pages 151–161. IEEE, May 2009.

[41] C. Stirling. *Modal and Temporal Properties of Processes*. Springer Verlag, New York, Berlin, Heidelberg, 2001.

[42] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award Paper.

[43] Yang Xu and Kenneth S. Stevens. Automatic Synthesis of Computation Interference Constraints for Relative Timing. In *26th International Conference on Computer Design*, pages 16–22. IEEE, Oct. 2009.

[44] Eslam Yahya and Marc Renaudin. QDI Latches Characteristics and Asynchronous Linear-Pipeline Performance Analysis. In *Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation*, Lecture Notes in Computer Science, pages 583–592. Springer, 2006.

[45] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. *Formal Methods in System Design*, 9:139–188, 1996.

[46] Kenneth Y. Yun, Peter A. Beerel, and Julio Arceo. High-performance asynchronous pipeline circuits. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 17–28. IEEE Computer Society Press, March 1996.