DATA-DRIVEN HANDSHAKE CIRCUIT SYNTHESIS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2007

By
Samuel M Taylor
School of Computer Science

Contents

List of Figures

List of Tables

Abstract

This thesis describes a novel method of synthesising asynchronous circuits based upon the Handshake Circuit paradigm used in the Balsa synthesis system but employing a data-driven style, rather than the control-driven style of conventional Balsa. This approach attempts to combine the performance advantages of data-driven asynchronous design styles with the handshake circuit style of construction for synthesising large circuits.

The integration into the existing Balsa design flow of a compiler for descriptions written in a new data-driven language is described along with the implementation of a number of new handshake components to support the new style.

The method is demonstrated using a significant design example — a 32 bit microprocessor. This example shows that the data-driven circuit style does indeed provide better performance than conventional control-driven Balsa circuits. Some qualitative discussion on the relative merits of the new description language when compared with conventional Balsa is also presented.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

Any list such as this is inevitably incomplete and no doubt, before the ink is even dry, some names will occur to me that I shall regret failing to mention. Thanks to you all.

Thanks must go to my supervisor Doug Edwards and advisor Linda Brackenbury. Also to the members of the APT group, and particularly all those who have contributed to Balsa.

I must also express personal thanks to my friends Andrew Robinson, Ian Jason, Matt Horsnell and all the folks at Linton.

Finally, and despite the many shortcomings of his processor architecture, I must express my deepest gratitude to Luis Plana. Without his selfless assistance and support this thesis would never have been completed. Gracias mi amigo.

Chapter 1

Introduction

In recent years there has been increased interest in asynchronous or 'self-timed' techniques for digital circuit design [SF01, BJN99, BRD95]. In contrast to nearly all circuits currently designed, asynchronous circuits do not rely on the presence of a global clock. Many possible advantages are put forward for asynchronous techniques including the avoidance of problems related to clock distribution and clock skew [Fri01]. Additionally, increased modularity [SKC+99], increased robustness in the face of environmental and process variation [NS94, NSJ90, MBL+89], lower power consumption [GBvB+98, BBK+94], low electromagnetic noise [FGT+97, GBvB+98, PDF+98], improved security [MAC+02, PRB+03], and high performance [Bre06, MLM+97, WPS95, FGG98] are all claimed as possible advantages. There is perhaps more evidence to support some of these claims than others. It is not intended that this thesis should argue specifically for the advantages, or any particular advantage of asynchronous techniques over their synchronous counterpart.

The vast majority of synchronous circuits are synthesised partly or entirely using computer-aided design (CAD) tools. It is clear that if asynchronous techniques are to gain more widespread acceptance, then robust and efficient synthesis tools are a necessity. Balsa is one such tool, designed for high-level synthesis of asynchronous circuits from algorithmic language descriptions. Balsa has demonstrated that it is robust and flexible, and can be used for the rapid development of large designs. However, this speed and flexibility is achieved at the cost of performance in the resulting circuits. Competitive performance must be demonstrated before any other potential advantages of asynchronous techniques will be seriously considered by the synchronous design community.

This aim of this thesis is to contribute to knowledge in the field of synthesising large asynchronous circuits with the specific objective of improving performance. Area and power are not considered relevant factors though, on occasion, some small consideration is made where it was possible with minimal additional effort.

1.1 Asynchronous synthesis methods

Existing asynchronous synthesis methods may be broadly grouped into four categories. The first of these groups is restricted to the synthesis of small-scale asynchronous control circuits. Most of these methods use either Petri nets [CKK+97] or burst-mode machines [FNT+99] as specifications for asynchronous control circuits. This work in this thesis is aimed at the synthesis of large circuits inclusive of both control and datapath. Therefore, these controller synthesis methods are of limited interest although Petri nets are appropriated as a convenient method of specifying handshake component behaviour (see section 2.2).

The methods that target synthesis of large-scale circuits are described in the following three sections.

1.1.1 De-synchronisation based methods

This method involves converting conventional synchronous design descriptions into asynchronous designs [CKLS06, KL02]. Typically existing CAD tools are used for much of the datapath synthesis and asynchronous control synthesis tools are used to produce controllers that replace the global clock. This approach has the advantage that designers need little specialist knowledge of asynchronous techniques. A drawback is that by using a design targeted at a synchronous implementation, potential advantages of asynchronous techniques are not exploited. For example, concurrency is restricted to the synchronous pipeline structure and so the fine-grained concurrency possible in asynchronous design is not exploited. It is also difficult to exploit the possibility for asynchronous designs to use data-dependent delays instead of the worst-case delays of synchronous design.

1.1.2 CHP based methods

The CSP[Hoa85]–based Communicating Hardware Processes (CHP) language is the basis of some asynchronous synthesis systems [Mar90, RVR99, TAS]. These systems use manual or automatic program transformations to refine a design into a more concurrent version. The final program is then translated into a production-rule set which is used to generate a transistor implementation of the design.

The Caltech synthesis tools (CAST) have been used to produce some high performance circuits [MLM+97] but these rely on significant manual intervention in the synthesis flow to arrive at the most effective program transformations and also rely on the use of the PCHB (precharge half-buffer) circuit style. This circuit style is not widely used and requires a specialised cell library.

The automatic program transformations employed in CAST are not behaviour preserving and are only correct for designs that meet particular requirements. An inexperienced designer may struggle to understand and meet these requirements.

1.1.3 Macromodular based methods

The term macromodular originates from the Macromodules system developed at Washington University [Cla67]. This was a system of large rack-mounted modules that were physically connected by hand. Current macromodular systems are somewhat smaller but share the basic concept of composing small pre-designed modules to produce large systems. Two prominent asynchronous synthesis tools use the handshake circuit paradigm first proposed by Van Berkel [Ber93] as the intermediate representation for the asynchronous circuits compiled from the language Tangram[1]. The Balsa synthesis system is heavily based on Tangram and uses the same paradigm. Balsa offers a few different features to Tangram but largely differs only in small details. The work in this thesis is based on Balsa and so the remainder of this section will describe the synthesis method of Balsa in some detail although the description is equally valid for Tangram.

Balsa is a framework for high-level synthesis of asynchronous circuits. Balsa is also the name given to the main language in which circuit descriptions are written. These descriptions are compiled into networks of communicating handshake components called handshake circuits. Handshake circuits are an attractive paradigm as they offer a level of abstraction above any particular implementation style or technology. Handshake circuits exploit the modularity of asynchronous techniques in the synthesis of large-scale systems. Each handshake component is straightforward to construct in isolation. By composing the components, very large systems may be robustly constructed. The translation employed by Balsa from language description to handshake circuit is described as syntax-directed. This means the structure of the resulting circuit is based on the syntax of the source code. This provides the advantage that the resulting circuit may be optimised for power, area or performance at the

[1]latterly renamed as Haste

language level. The translation is also described as control-driven. The handshake circuit network features a control tree which mirrors the control flow of the language description. The overhead of this control-driven approach is a major factor in restricting the performance of this style of handshake circuit.

The work in this thesis is based upon the handshake circuit method of construction and the syntax-directed method of translation. In place of the control-driven approach, a novel style of handshake circuit is proposed, based much more on data-flow rather than control-flow. This approach is described as *data-driven*.

1.2 Aims of this research

The aim of this research is to improve the performance of large synthesised asynchronous circuits. The focus of the approach is on a handshake circuit representation of the circuit; that is to say, an abstract representation of the structure of the circuit which is independent of technologies, protocols, data-encodings or any other details of the actual circuit implementation. The problem of control overhead in the conventional control-driven style of handshake circuit synthesis is identified as a major obstacle to performance. Data-driven asynchronous design styles are much less prone to the problem of control overhead and so the approach of this research is to combine the benefits of a data-driven style with the convenience and flexibility of the handshake circuit paradigm which allows the robust synthesis of large circuits. To this end, the handshake circuit structures of the control-driven Balsa synthesis method have been examined and data-driven alternatives are proposed. To generate these structures, a data-driven description style is proposed and a compiler has been developed to compile these description into a handshake circuit representation. This compiler is integrated into the Balsa design flow enabling the use of existing Balsa tools for moving from the handshake circuit representation to a gate-level circuit.

The benefits of the new style are successfully demonstrated by the manual translation of an existing high performance Balsa design of significant size and complexity directly into the data-driven style.

1.3 Contributions of this thesis

The contributions made by this thesis can be summarised as follows:

- A novel synthesis method for asynchronous circuits combining the performance benefits of data-driven design styles with the handshake circuit paradigm for constructing large circuits.
- A hardware description language that is specifically tailored for syntax-directed translation into the data-driven circuit style.
- Demonstration of the use of the synthesis method in a significant design example (a microprocessor).
- Analysis of the performance improvements gained by using the data-driven method over conventional control-driven handshake circuit synthesis.

1.4 Thesis structure

The remainder of this thesis is divided into five chapters as follows:

Chapter 2 gives background information on asynchronous design fundamentals and on the existing Balsa synthesis system.

Chapter 3 presents the main work of this thesis: a data-driven handshake circuit style and language from which this circuit style is compiled. This is preceded by a discussion of the control overhead of conventional Balsa circuits and the manner in which the data-driven style is less acutely affected by this problem.

Chapter 4 contains a range of information and ideas on the implementation and usage of the data-driven style. Some ideas for future work are briefly discussed.

Chapter 5 describes the implementation of a data-driven implementation of the nanoSpa processor. This implementation is compared with the control-driven original in an effort to evaluate the strengths and weaknesses of the proposed data-driven approach.

Chapter 6 summarises the work presented herein and offers suggestions for future work.

A number of appendices offer supplementary information of a more detailed nature:

Appendix A gives the grammar of the data-driven language.

Appendix B gives the implementations of new handshake components introduced to the Balsa component set to support the data-driven style.

Appendix C gives code for selected modules of the data-driven nanoSpa description.

Finally, appendix D gives very brief descriptions of the handshake components in the Balsa component set. These descriptions are intended to act as a reminder of the component behaviours.

1.5 Publications

The author has contributed to the following papers while conducting the work described in this thesis.

- Luis Plana, Doug Edwards, Sam Taylor, Luis Tarazona and Andrew Bardsley. Performance-driven syntax-directed synthesis of asynchronous processors. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES)*, September 2007.
- Luis A. Plana, Sam Taylor, and Doug Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *Proc. International Conf. Computer Design (ICCD)*, pages 703-710. IEEE Computer Society Press, October 2005
- Sam Taylor and Doug Edwards. Control localisation as a means of improving handshake circuit performance. In *Seventeenth UK Asynchronous Forum*, pages 1-4.

Chapter 2

Background

This chapter contains the background information upon which the work described later in the thesis is based. The principles of asynchronous handshaking are briefly described followed by more specific background on the Handshake Circuit paradigm and the Balsa synthesis system.

2.1 Handshaking

The basis of nearly all asynchronous design techniques is the concept of handshaking to provide synchronisation between communicating modules. This synchronisation is frequently used to facilitate the transfer of data between the modules. Multifarious technologies and protocols exist for the implementation of these handshakes but conceptually all rely on the exchange of a request and acknowledge signal between communicating modules. One module initiates a handshake by sending a request (req) signal and the other, when ready, responds with an acknowledge (ack). For systems that wish to abstract away the detail of the implementation of the handshake it is common to combine the control for signalling the req and ack, together with any accompanying means of transferring data, in a single conceptual unit known as a *channel*. The channel therefore forms a point-to-point link between the two modules that it

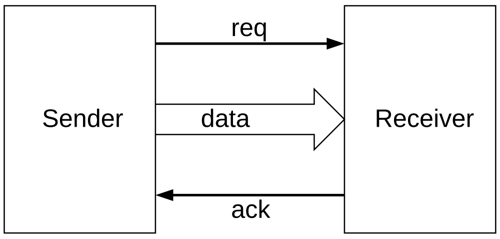


Figure 2.1: Push data channel

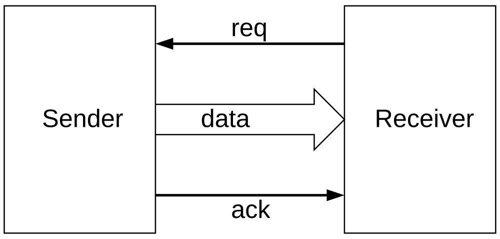


Figure 2.2: Pull data channel

connects, either for the purpose of making data transfers, or simply to allow the modules to synchronise.

Data channels can be divided into two types: where the data moves in the same direction as the request and where the data moves in the same direction as the acknowledge (see figures 2.1 and 2.2). These two possibilities are named *push* and *pull* respectively.

2.1.1 Handshake protocols

Much of this thesis is concerned with the handshake circuit structures of Balsa and of the new data-driven style. The handshake circuit paradigm provides a level of abstraction that is neutral as to which technologies and handshake protocols may be used to implement the final circuit. However, in order to implement the handshake components and produce real circuits, issues of protocol must be addressed.

The two most widely used handshaking protocols are known as *two-phase*

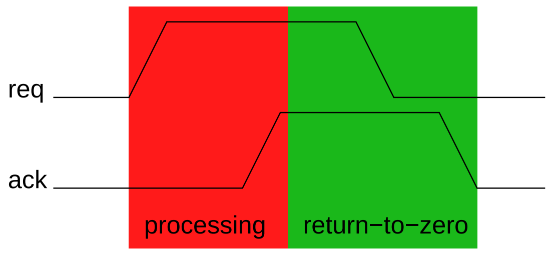


Figure 2.3: Four-phase handshake protocol

or transition signalling and *four-phase* or level-sensitive. A two-phase protocol is conceptually more straightforward in that each handshake consists of a single request event and single acknowledge event. For example, two wires may be used, one for the request and one for the acknowledge. A transition on the request wire indicates a request and a transition on the acknowledge wire indicates an acknowledge. A two-phase scheme is perhaps most notably employed in the Micropipelines style [Sut89]. Unfortunately, it is often difficult to implement efficient circuits using a two-phase protocol due to the complexity of detecting transitions on wires compared with detecting levels. This is particularly true when using a multiple-wire encoding scheme (see the next section) although it has been applied in some applications [FES00].

Four-phase handshaking uses the level of wires to indicate signals so, for example, a high request wire indicates a request and a high acknowledge wire indicates an acknowledge. Each handshake must then return the request and acknowledge to their original state before the next handshake may begin. The four-phase protocol can therefore be considered as having two sub-phases: the *processing* phase and the *return-to-zero* (RTZ) phase (see figure 2.3).

2.1.2 Data encoding

Two broad categories of design style are commonly employed in asynchronous design. These are known as *bundled-data* and *delay-insensitive*. Bundled-data designs nearly always employ traditional single-rail data encoding where one

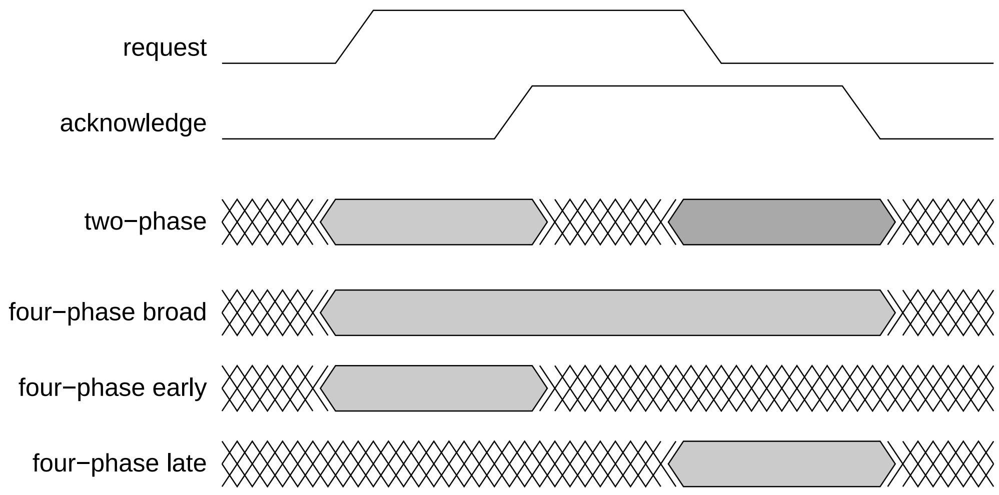


Figure 2.4: Push data validity protocols

wire represents each bit. Most commonly, additional request and acknowledge wires are bundled with the data although techniques exist that use a single wire for both request and acknowledge [BB96]. When using bundled data it is necessary to define a data validity protocol that determines where in the handshake the data is valid. For two-phase there is little option as to the data validity protocol, as there are only two events. For four-phase there are several options for the period that data must be valid. Figures 2.4 and 2.5 show the common data-validity protocols for push and pull channels respectively. The standard bundled-data Balsa back-end uses the four-phase broad protocol for push channels. For pull channels the broad protocol is less convenient as the data validity must be maintained until the request goes up in the following handshake. For this reason, the reduced broad protocol is used for pull channels. In theory, there is little to distinguish the reduced broad and early protocols as once the receiver has lowered its request then the acknowledge could be lowered and data changed immediately. In practice, the ack and data will not change immediately and so it is possible to assume a small extra period of validity will exist after the request is lowered and exploit this in component implementations.

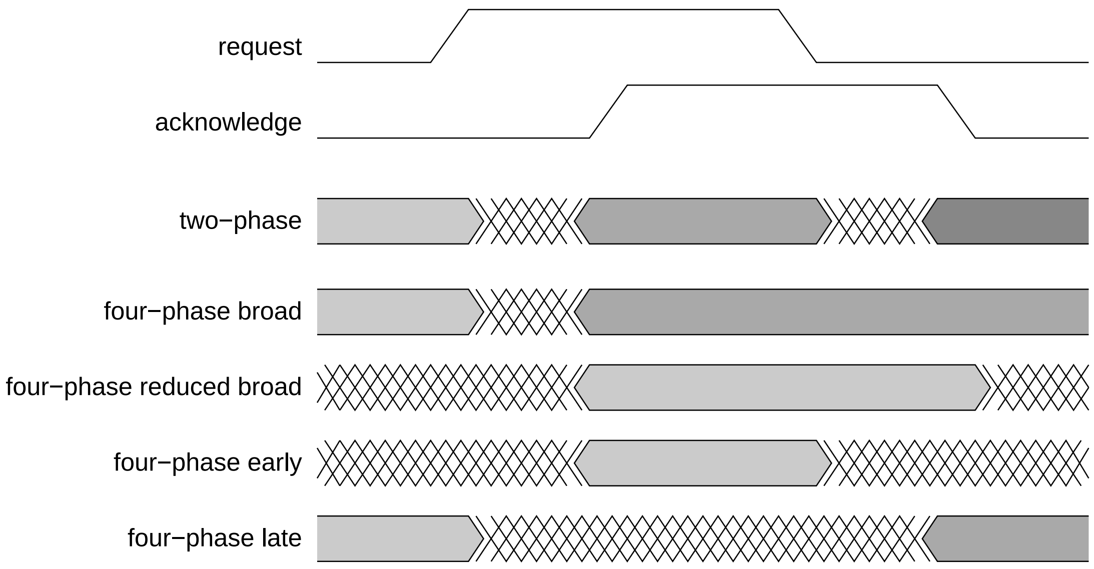


Figure 2.5: Pull data validity protocols

Delay-insensitive systems use multiple wire data encodings; most prevalent of these is the dual-rail encoding where two wires are used to encode each bit. A delay-insensitive encoding allows the data to signal when it is valid and the data itself signals either the request (for push channels) or acknowledge (for pull channels). One additional wire is used for the signal that is not encoded with the data. See figure 2.6 for an example of the handshakes for one-bit dual-rail push and pull channels using a four-phase RTZ protocol. One of the two wires carrying the data is used in each handshake. One wire indicates a zero is being sent (req0 or ack0) and the other indicates a one is being sent (req1 or ack1). It is uncommon to use an extra wire to request or acknowledge every bit in a data channel carrying multiple bits. Instead a single extra wire is used for the channel and it is then necessary to use *completion detection* to check that all the bits have arrived and have gone away before transitioning the request or acknowledge signal.

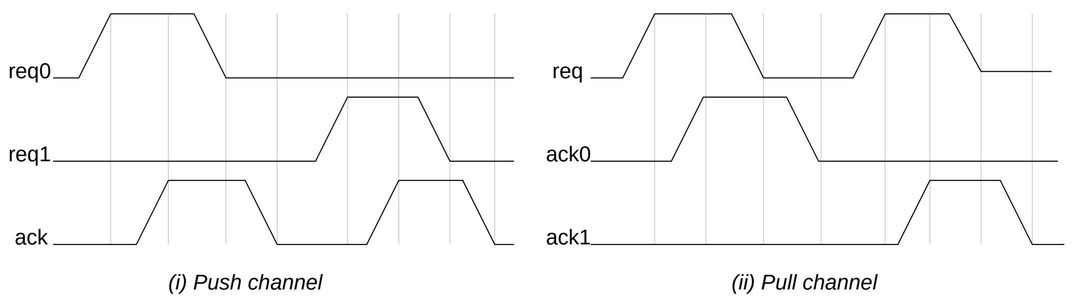


Figure 2.6: Four-phase dual-rail handshakes

2.2 Signal Transition Graphs

Signal Transition Graphs (STGs) are a specific form of Petri net — a well established formalism for modelling concurrent systems. STGs are used as an input format to some asynchronous synthesis systems such as Petrify [CKK+97]. These methods rely on state-space exploration and so are only suitable for the synthesis of small controllers as for large designs the state-space quickly explodes to an unmanageable size. In this thesis STGs will be employed as a convenient method of specifying the behaviour of handshake components.

A Petri net is made up of places and transitions connected by directed arcs. Transitions indicate events in the system; in the case of circuit design these are signal transitions. Places can hold a number of tokens. Each transition may have a number of inputs and outputs which are the places that are connected to and from the transition. The operation of the Petri net proceeds by firing transitions. Transitions are enabled when all of their inputs have at least one token. When a transition fires a token is removed from each of its inputs and a token added to each of its outputs. This may then enable further transitions. The number of tokens at each place in the system at any given time is called the *marking*.

An STG is a Petri net with two specific restrictions. An STG must be one-bounded which means that at all times only one token is allowed at each

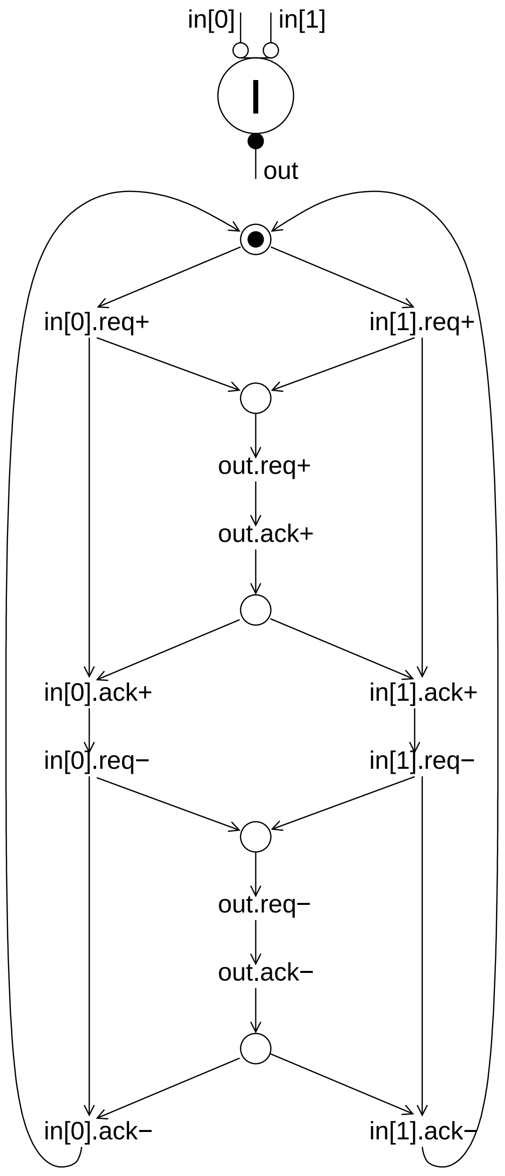


Figure 2.7: Call STG

place. Only inputs may be used as transitions where choice is involved and the signal transitions represented by such inputs must be mutually exclusive. Where choice occurs, a single place has multiple transitions that could fire and consume a token. The signal transitions in the implementation must provide means of deciding which transition should fire.

Figure 2.7 shows the STG for the Balsa Call component in order to demonstrate the graphical representation of an STG. Places are drawn as circles but are usually omitted between transitions that are directly connected through a single place. A line connecting two transitions directly therefore has an implicit place in the middle. The initial marking of the circuit is shown using a filled circle either drawn within a place or next to an arc that has an implicit place. Note particularly the representation of choice between the input requests. Either req+ transition may consume the same token. As the input requests to the component are input signals and are mutually exclusive, it is possible to choose the correct transition.

The figure also shows the convention that will be used in STGs and circuit diagrams throughout this thesis for showing the expansion of channels into their constituent parts. For example, the channel **out** is expanded to out.req and out.ack. The upward transition on out.req is represented by out.req+ and the downward transition by out.req-.

2.3 Balsa design flow

An overview of the Balsa design flow is shown in figure 2.8. There are two synthesis stages in this flow that are handled by tools in the Balsa framework. The first is the compilation from a Balsa code description into the handshake circuit representation using the Balsa compiler. This is frequently called the *front-end*. The breeze format is in essence simply a list of channels and handshake components. The compilation is modular and each procedure in the

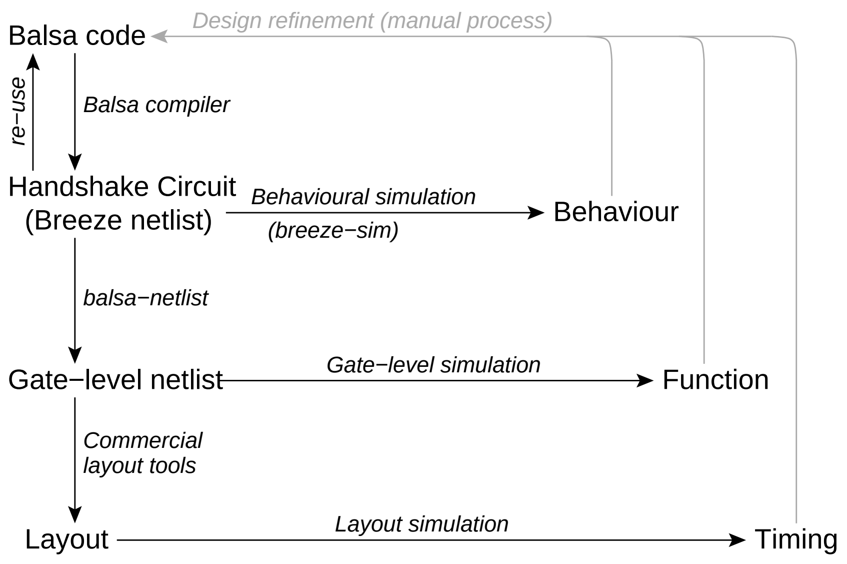


Figure 2.8: Balsa design flow

language is compiled into a breeze 'part'. The compiler will import breeze descriptions when a procedure from another file is instantiated in order to ensure the interface is correctly generated.

The second stage (or *back-end*) is the replacement of each handshake component with a gate-level implementation in the chosen design style and technology performed by the balsa-netlist tool. The back-end produces a gate-level netlist which can be processed by commercial place and route tools for layout to silicon or possibly to an FPGA.

This thesis is mainly concerned with the first synthesis stage — from language description into handshake circuit. The conventional Balsa compilation process is described in some detail in the following section. This thesis describes a new compilation approach that complements the existing Balsa compiler in this design flow. The additions to the flow will be discussed in section 4.1 on page 115.

To implement the new data-driven style, many existing handshake component implementations are re-used, but in addition several new components are added to the Balsa handshake component set. Section 2.6 gives some background on common techniques that have been employed in implementing these components. The next two sections give more detail on the handshake circuit paradigm and the front-end compilation of Balsa source descriptions into handshake circuits.

2.4 Handshake Circuits and Balsa

The handshake circuit is a network of small components connected by channels. The network is generated by a compiler (frequently called the front-end) that translates a high-level language description into a handshake component network. The compilation involves converting each language feature into a small structure of handshake components which implements that feature and composing these smaller structures based on the syntax of the written description. This approach is often described as 'syntax-directed' or 'transparent' compilation as there is a fixed relationship between language descriptions and the circuit that they generate. Small changes in the description will produce small and predictable changes in the resulting circuit. Furthermore the direct compilation allows the construction of large-scale designs and is one of only a few proven methods for doing so in an asynchronous style.

2.4.1 Handshake circuit diagrams

Figure 2.9 illustrates the diagrammatic representation of a small handshake circuit. Handshake components are usually rendered as a circle containing a symbol that indicates the type of component. Each component has one or more ports to which channels are connected. The *sense* of the port indicates whether it initiates communication (sends the request) or responds to communication (sends the acknowledge). The active port, drawn as a small filled circle, sends

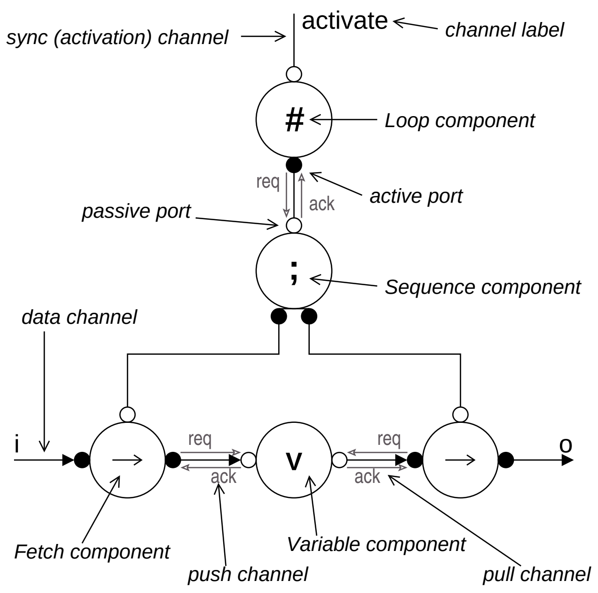


Figure 2.9: Handshake circuit diagram

the request and the passive port, drawn as a small open circle sends the acknowledge. (Example requests and acknowledges are shown in the figure but are not usually shown.) A channel always connects an active port from one component to a passive port from another. Channels are represented by lines; arrows are added to the lines to indicate data channels where the direction of the arrow indicates the direction in which the data flows. Data channels can be further divided into two types: push channels where data flows in the same direction as the request and pull channels where data flows in the direction of the acknowledge.

Channels without data are called *sync* channels, or often *activation* channels, as they are in the most part used for the purposes of activating components in the circuit.

2.5 Balsa language and compilation

The Balsa language is fully described in the Balsa Manual[EBJ$^+$06]. A brief overview will be presented here with much detailed information that is not directly relevant to this thesis being omitted. Accompanying the language descriptions are examples of the handshake circuit structures produced when the language constructs are compiled. Copious information on the compilation and handshake components employed therein may be found in [BE97, Bar00, PTE05].

Balsa descriptions are divided into procedures. Each procedure has an implicit activation port that activates the circuit described within the procedure. In addition, each procedure may have any number of input, output, or sync (non data carrying) ports. These ports are the external interface to the procedure. From within the procedure, ports are used as if they were channels but they are read-only for input ports and write-only for output ports. In addition to ports, procedures provide scope for local channel and variable declarations. Channels in this context are a language feature and do not normally correspond to an individual channel in the handshake circuit.

The body of a procedure consists of commands, composed using control structures. Each command or structure is compiled into a small network of handshake components with an activation channel that is used to control when the command operates. The compilation connects the small network to the overall handshake circuit network by attaching the activation, and any input and output channels to the appropriate points in the overall handshake circuit.

2.5.1 Data types

Balsa supports global and local type and constant declarations. Basic numeric types in Balsa can all be considered as bit vectors of a given width and can be signed or unsigned. Array, enumeration and record types are supported.

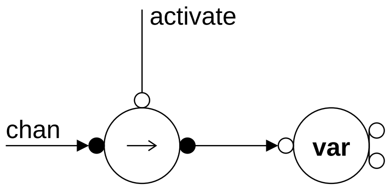


Figure 2.10: Balsa channel read (and write into a variable)

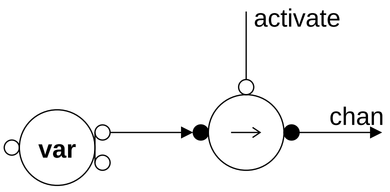


Figure 2.11: Balsa channel write (from a variable)

Ports, channels and variables are all declared as having a specific type.

Details of data types will in general be omitted from examples in this thesis as the primary interest is in the structures of the handshake circuits. The width of data involved is rarely significant to the structures, which are applied in the same fashion to data of any given width.

2.5.2 Basic commands

There are four basic commands in Balsa: channel reads, channel writes, continue and halt.

A channel read is used to write the data from a channel into a variable. It is written using the `->` operator, e.g. `chan -> var`. It is compiled to a *Fetch* component as shown in figure 2.10. A channel read may also be used to write the data to another channel in place of the variable (e.g. `chan1 -> chan2`).

A channel write is used to read data from a variable and output it to a channel. It is written using the `<-` operator, e.g. `chan <- var`. It is also compiled to a Fetch component as shown in figure 2.11.

The continue command is used to perform no operation and is compiled to

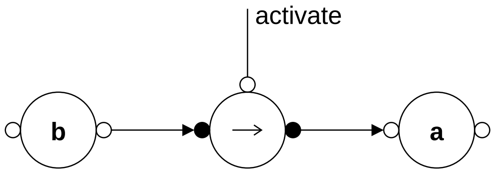


Figure 2.12: Balsa assignment

a component that simply acknowledges any activation request it receives. The halt command never acknowledges an activation so that the circuit deadlocks at the point where the halt occurs. (Other independent parts of the circuit may continue operating so the entire circuit may not deadlock.)

The assignment command (:=) is also available in Balsa. It is in reality a compound command made up of a channel write and a channel read but the channel is implicit. For example, the following two code fragments are equivalent:

```
variable a, b

-- this assignment is implicitly...
a := b

-- the same as this...
channel c

c <- a || c -> b
```

The handshake circuit generated for both fragments is the same and, due to a small optimisation, does not use two Fetch components but only a single Fetch component as shown in figure 2.12.

2.5.3 Parallel and Sequence control

The basic commands may be composed using the concur (||) and sequence (;) operators to form compound commands. A single-place buffer may be described by composing a channel read and write in sequence as shown below.

```
procedure buf (input i : byte; output o : byte) is
    variable V : byte
begin
    loop
        i -> V ;
        o <- V
    end
end
```

Figure 2.9 shows the handshake circuit for this example. The *Sequence* component sequentially activates its active ports so the channel read is activated first and upon its completion the channel write is activated.

The parallel operator is similar but produces a *Concur* component in place of the Sequence. This component activates its active ports in parallel and waits for them to complete before acknowledging on its passive port.

2.5.4 Conditional control

Conditional control is provided by the `case` and `if` structures. If is fundamentally the same as case so this discussion will use case as an example. The case construct is written as follows:

```
case <expression> of
  <guard0> then <command0>
| <guard1> then <command1>
.
.
.
  <guardN> then <commandN>
else
    <else_command>
end
```

The guards may consist of a comma-separated list and must be resolvable at compile time. They must also be disjoint in that no value may be matched by more than one guard. The else clause is used to match any values not covered by the guards and is optional in Balsa even if the guards do not exhaust all

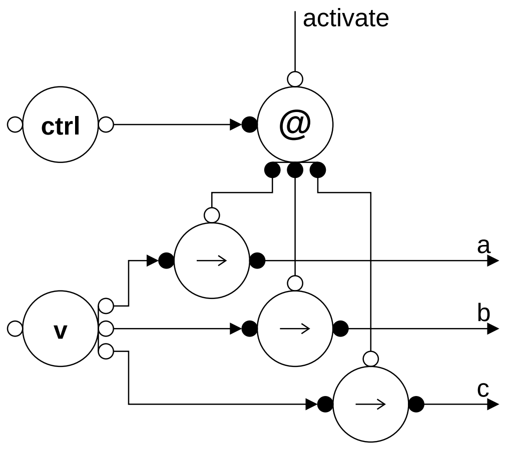


Figure 2.13: Balsa case example

possible values of the expression. If the else is omitted then the behaviour is the same as if the body of else were a continue command.

The case command is compiled to the *Case* component. The Case component has a passive activation port which when activated initiates the evaluation of the expression. The result of the expression is used to determine which of the output activation ports will be activated. For example the following code uses a case structure to select which of three channels (a, b, c) the data from variable v should be written to. This is compiled into the handshake circuit shown in figure 2.13.

```
case ctrl of
  0 then
    a <- v
| 1 then
    b <- v
| 2 then
    c <- v
end
```

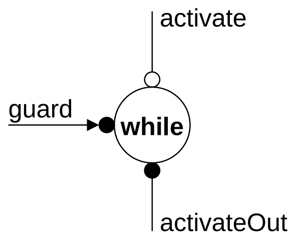


Figure 2.14: Balsa While component

2.5.5 Iterative control

Balsa has two iterative structures, `loop` and `while`. Loop is very straightforward as it simply repeats the command given in its body indefinitely. It is compiled to the *Loop* (#) component as shown in figure 2.9. This component, upon receiving an activation, repeatedly handshakes on its output activation port and never acknowledges the input.

The second structure, `while`, provides finite iteration as found in most imperative programming languages. It is written as follows:

```
loop while <expression> then
    <command>
end
```

The result of the expression must be a single bit. This while loop is compiled using the *While* component (figure 2.14). Upon activation the component pulls on its guard port to get the result of evaluating the expression. If the result is 1 then the output activation is sent to activate the body of the loop and when this is completed another guard is fetched. When the result of the guard is 0 then the while loop terminates and acknowledges its activation.

2.5.6 Input enclosure

Input enclosure allows the handshake on one or more input channels to be held open while a command is activated. This allows the value on the channel(s) to

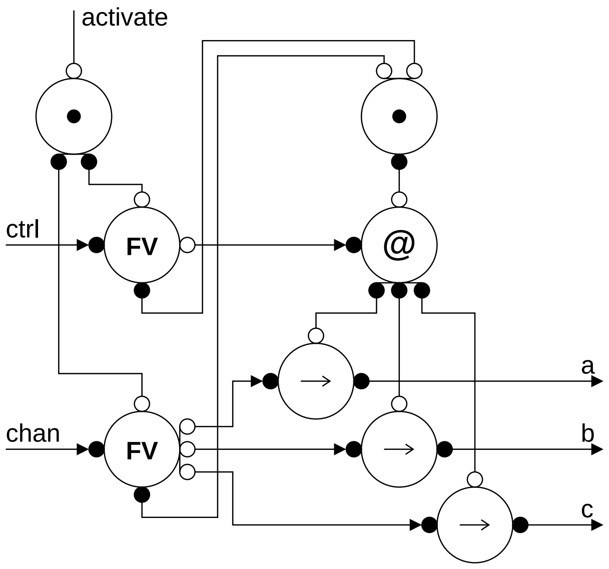


Figure 2.15: Balsa input enclosure example

be read as many times as desired by the enclosed command. The input channel(s) will only be released when the command has completed. For example the case construct shown in the conditional control section above may be used with channels instead of variables by using input enclosure as follows:

```
chan, ctrl -> then
    case ctrl of
      0 then
        a <- chan
    | 1 then
        b <- chan
    | 2 then
        c <- chan
    end
end
```

Figure 2.15 shows the handshake circuit produced by this code. The *False-Variable* (FV) component is used to implement the enclosure. Upon receiving an activation, the FV pulls the data from the required channel but it does not complete the handshake on this port. It then activates its 'signal' port to initiate the enclosed command. In this example there are two enclosed inputs so the activation is forked to two FVs and the signals are then synchronised to ensure both inputs have arrived before activating the enclosed case command. The FV provides passive read ports, much like a Variable component, on which the data can be read zero, one or many times. Within the enclosed command the channel is treated as if it were a variable. When the enclosed command completes the FVs complete the handshake on the input channels and acknowledge their activations.

2.5.7 Arbitration

The arbitrate construct is used to implement conditional control based upon the arrival of communications on input channels. The syntax is as follows:

```
arbitrate <list of channels> then
    <command>
| <list of channels> then
    <command>
end
```

Two guards comprising a lists of channels are provided. If every channel in one of the lists have a communication pending, then that guard is true and the command for that list will be activated. If both guards become true very closely in time then it may not be possible to determine which occurred first. In this case an arbitrary decision is made. The command that is activated is enclosed by its input channels as discussed in the previous section, allowing any data on those channels to be read by the command.

2.5.8 Data processing

Balsa features a number of operators that may be used to build expressions in the language. A number of operators may be compiled into hardware but several others are provided that may be used with compile-time constants but have no 'run-time' implementation.

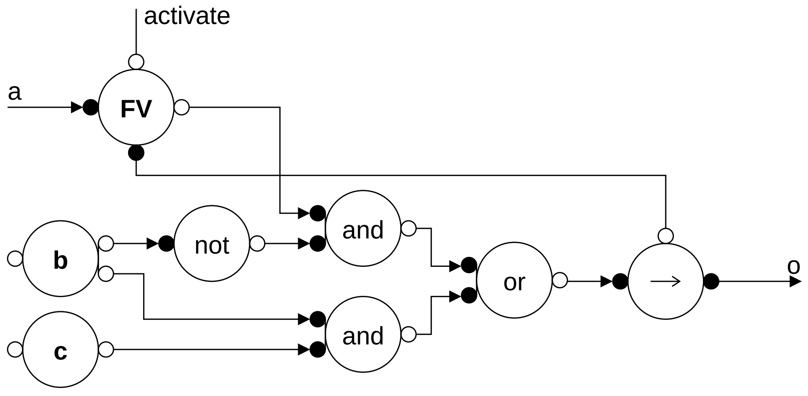


Figure 2.16: Balsa data processing example

Expressions are used with channel write commands to generate data processing logic. For example the following code produces a tree of data processing components as shown in figure 2.16 and a Fetch component to initiate the processing.

```
a -> then
    c <- a and not b or b and c
end
```

Expressions are also used as the input to case and while structures where the Case and While components are the initiators in place of Fetch.

Note how the data processing structure is always a pull structure and, as a consequence of this, a single activation can be used to initiate the pulling of the data through the tree of data processing components.

2.5.9 Miscellaneous connection components

The final few components to be mentioned do not correspond to any particular language structure but are used where multiple connections are made to a particular channel or variable. It has already been implied that the Variable component has a parameterisable number of read ports allowing the variable to be read from multiple locations in the code. Similarly the FalseVariable is

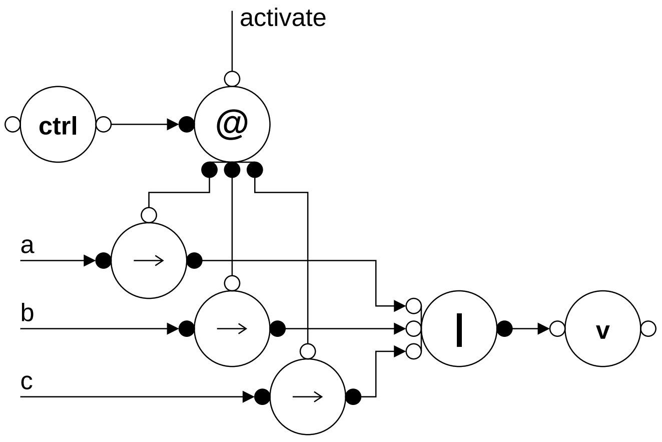


Figure 2.17: Balsa variable write from multiple possible sources

used to allow a channel to be read from multiple places. A variable and channel may also be written from multiple locations, providing they do not both attempt to perform a write concurrently. The *CallMux* component is used to merge multiple writes to a single channel or to a channel that connects to the write port of a variable. In the following code, a case statement is used to pick one channel (a, b or c) to write to variable v. The handshake circuit for this code is given in figure 2.17 which shows how the CallMux is used to merge the three possible write sources to the single write input to the Variable component.

```
case ctrl of
  0 then
    a -> v
| 1 then
    b -> v
| 2 then
    c -> v
end
```

The Fork component is sometimes used in place of Concur to fork an activation to two components where the overhead of a Concur is unnecessary. In a four-phase protocol a Concur component allows independent return-to-zero phases on each of its outputs whereas the Fork component synchronises

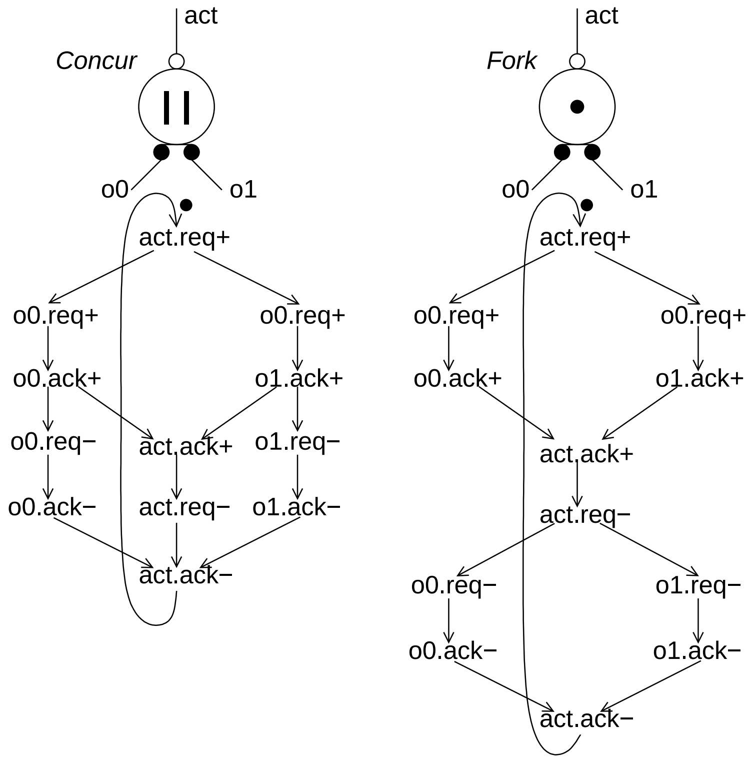


Figure 2.18: STG comparison of Fork and Concur

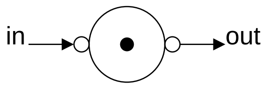


Figure 2.19: PassivatorPush component

following the processing phase of all the outputs before proceeding with the return-to-zero phase of all the outputs. Figure 2.18 illustrates the distinction between the two components by means of STGs. An example of the use of Fork is shown in figure 2.15 where it is used to fork the activation to each FalseVariable used for the handshake enclosure. As the two sides of the fork are symmetric, there is little advantage to be gained from having an independent return-to-zero phase in this instance.

The *Sync* component may also be encountered, often nearby a Fork as in figure 2.15. The Sync component is used to join several activations to produce a single activation that is triggered when all of the input activations have arrived.

The *Call* component (see figure 2.7) is used to merge several activations. An activation on any of the inputs is propagated to the output. The inputs must be mutually exclusive.

It can be seen in this section, that Balsa generates circuits with active inputs and active outputs. It is, of course, frequently necessary to connect an output from one process to the input of another. The *PassivatorPush* component (figure 2.19) is the used to accomplish this. PassivatorPush has a passive input and a passive output allowing data to be transferred from a push channel to a pull channel.

2.6 Implementing handshake components

The handshake circuit representation is independent of any particular implementation style or technology. However, in order to get meaningful results it is necessary to produce real circuits and so a particular implementation style

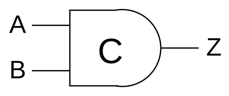


Figure 2.20: C-element with two inputs

A	B	Z	Z'
0	0	X	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	X	1

Table 2.1: C-element behaviour

must be selected. Tangram was originally proposed with a dual-rail implementation but a single-rail back-end is now commonly used [Pee96]. Balsa has several available back-end styles but the most commonly used are a four-phase broad bundled-data style and a dual-rail style. The broad bundled-data style in fact uses a broad protocol for push channels but a reduced broad protocol for pull channels. These two styles will be used in this thesis.

This remainder of this section introduces the asynchronous circuit elements that are used in the implementation of Balsa handshake components.

2.6.1 Control elements

C-element

The Muller C-element [Mul62] is a ubiquitous asynchronous component. It is sometimes considered as performing an **and** of signal transitions rather than of logic levels as in a regular and gate. The output of the C-element only transitions when all of the inputs have reached the same logic level.

The symbol for the C-element is shown in figure 2.20 and the behaviour is show in table 2.1. C-elements are often used for synchronising signals. For

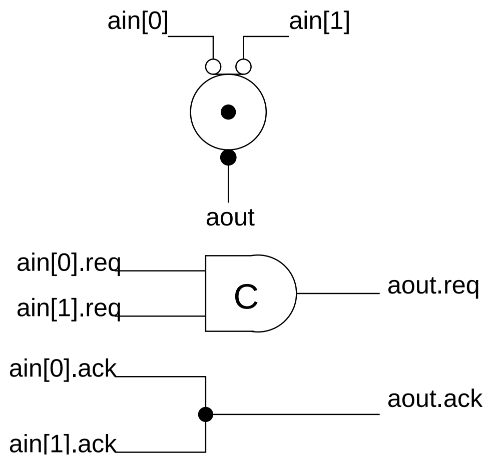


Figure 2.21: Sync component implementation

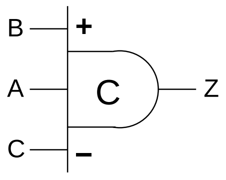


Figure 2.22: Asymmetric C-element

example, the Sync component described in section 2.5.9 may be implemented using a C-element as shown in figure 2.21.

The basic C-element can be extended by adding inputs that only affect transitions in one direction. Figure 2.22 shows an example asymmetric C-element and table 2.2 shows its behaviour. Inputs connected to the plus (+) symbol must be high for the output to transition from low to high while inputs connected to the minus (–) symbol must be low for the output to transition from high to low.

S-element

The S-element is a common component found in several handshake components. If the inputs and outputs are connected to the request and acknowledge

A	B	C	Z	Z’
0	X	0	X	0
0	X	1	0	0
0	X	1	1	1
1	0	X	0	0
1	0	X	1	1
1	1	X	X	1

Table 2.2: Asymmetric C-element behaviour

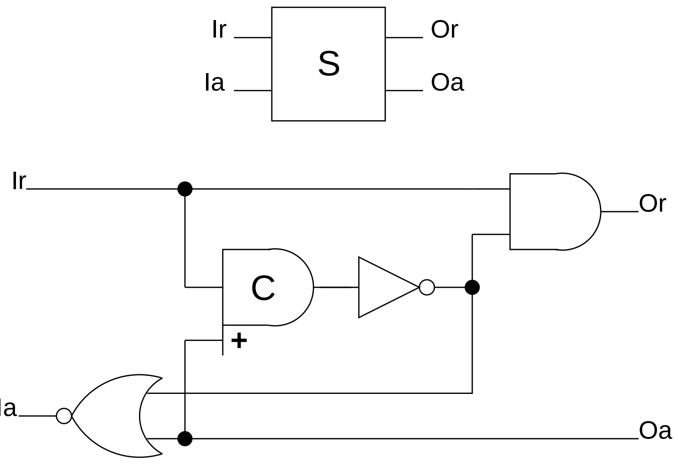


Figure 2.23: S-element implementation

signals of channels then it implements *handshake enclosure*; that is the handshake on one side (the output handshake) of the component occurs within the handshake of the other (input handshake) side. See the STG in figure 2.24 and the implementation in figure 2.23.

For example, consider the control of the bundled data FalseVariable component used to implement input enclosure as described in section 2.5.6. The handshake on the write port of the FV must enclose the handshake on the signal port. This is accomplished as shown in figure 2.25.

T-element

The S-element was originally used exclusively in Balsa component implementations. In later revisions it was realised that often the S-element is more

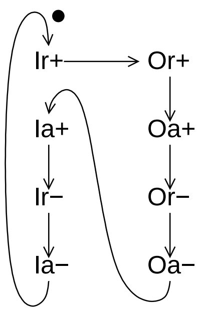


Figure 2.24: S-element STG

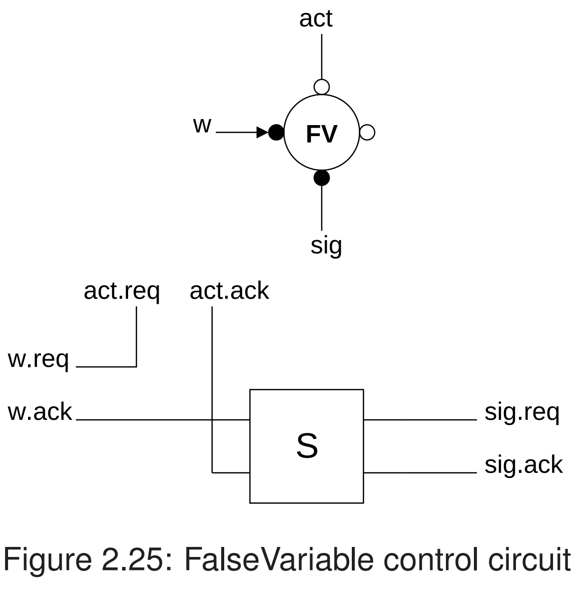


Figure 2.25: FalseVariable control circuit

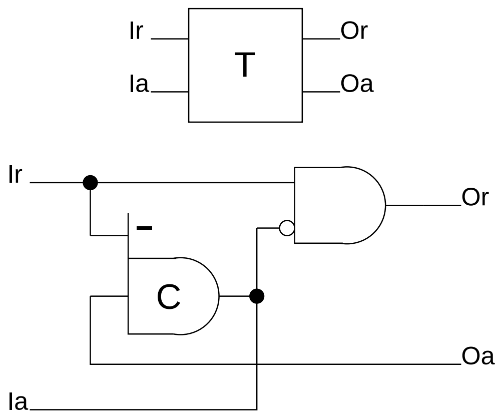


Figure 2.26: T-element implementation

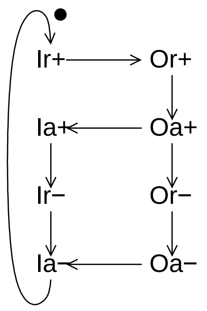


Figure 2.27: T-element STG

sequential than necessary [PTE05]. The T-element [NUK^{+}94, KPWK02] implements a form of enclosure similar to the S-element but the return-to-zero phases of the handshakes are overlapped allowing more concurrent operation. The standard FalseVariable design now uses a T-element in place of the S-element shown in figure 2.25.

2.6.2 Dual-rail elements

It is often convenient to generalise a variable width data bundle by drawing the entire bundle as a single arrow. A few symbols are used to show how this data is processed.

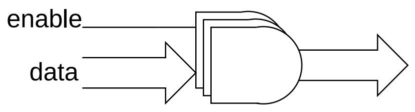


Figure 2.28: Read Port

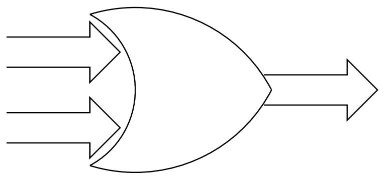


Figure 2.29: Merge

Read port

The symbol in figure 2.28 is used to represent a set of **and** gates, one for each wire in the data. The enable signal is distributed to all the **and** gates allowing the output of data to be controlled.

Merge

The symbol in figure 2.29 is used to represent a merging of multiple bundles of the same width. The corresponding wires from each bundle are passed through an **or** gate. In dual-rail it is imperative that data is not present on more than one input.

Decode

Decode (figure 2.30) is used to convert data into a one-hot code based on the value of the input data. In Balsa, this is achieved by specifying the values of the input data that correspond to each output wire. The specification is passed to the logic minimiser Espresso [BSVMH84, Esp] to generate an efficient circuit implementation.

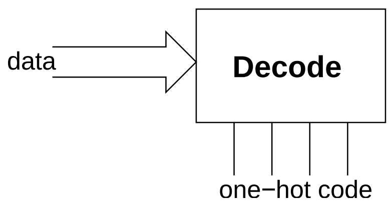


Figure 2.30: Decode

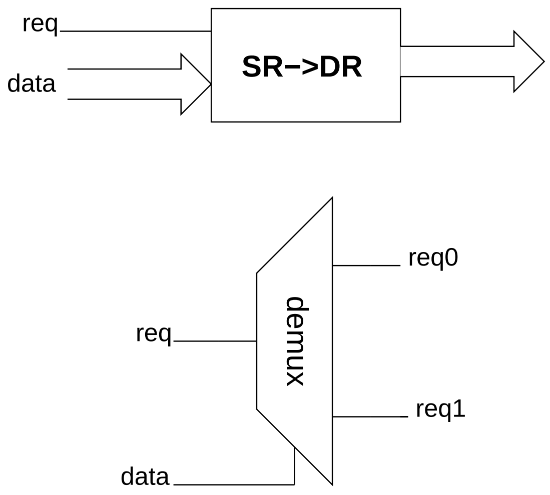


Figure 2.31: Single to dual-rail converter

Single to dual-rail converter

This element converts single-rail data to dual-rail. This is sometimes useful, for example in order to provide a dual-rail input to the decode element. Figure 2.31 shows the implementation of the converter for each data bit.

2.6.3 Completion Detection

The arrival of a single bit in dual-rail encoding is detected by an **or** of the two data wires. To detect the completion of a wider bundle of data a C-element is used to combine the completion of each individual bit.

Completion detection is shown as a single element on schematics as shown in figure 2.32.

Sometimes it is not necessary to use completion detection for the arrival of

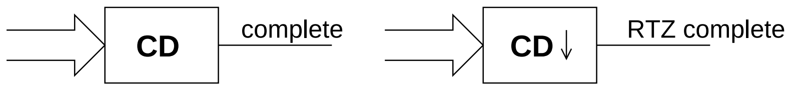


Figure 2.32: Completion Detection

data but only to check that it has returned to zero. This allows the implementation to be simplified as **or** gates may be used in place of the C-elements. A downward pointing arrow is added to the CD element to indicate this variety of completion detection.

2.7 What does ‘performance’ mean in an asynchronous circuit

It is important to distinguish what is meant by the term ‘performance’ which will be used throughout this thesis. In general the term is used to mean throughput rather than latency, but of course these are not unrelated.

It is easy to determine the performance of a synchronous circuit. The latency of the worst-case delay (critical path) through the logic of the circuit determines the fastest possible clock speed. Or in practice, a target clock speed is chosen and used to determine the maximum latency of the critical path. A design is usually partitioned into pipeline stages in order to reduce the latency of the critical path. The completion of every stage is determined by the global clock and so every stage is restricted to operating at the speed of the slowest. The throughput is determined by the clock cycle.

In an asynchronous system there is no external clock to determine the completion of logic. Instead, each stage is free to individually determine when it completes. Each stage may have a fixed delay but, more often, the delay is data-dependent – the delay varies depending upon the operation being performed. In a synchronous design, each stage produces a result on a clock signal and begins the next operation on that same signal. In an asynchronous system the handshake signalling means that the cycle time is not determined solely by the latency of producing a result but also of acknowledging that result and resetting the stage back to a state where the next operation can begin.

In this thesis the throughput of an asynchronous circuit will usually be determined by measuring the average cycle time of the circuit while repeatedly performing an operation. The cycle time is the delay between producing consecutive results in the repeated operation. For modules whose operation can vary depending on data, a number of different operations will be performed to determine the variation in cycle time.

Chapter 3

Data-driven Circuit Style

3.1 Control overhead

Section 2.4 gave a brief introduction to the compilation of Balsa into handshake circuits. The resulting circuit may be roughly split into two sections: control and datapath. The datapath consists of Variable components, data processing structures and data channels. The control consists of a tree of control components connected with sync channels, which direct the movement of data around the datapath by activating interface components such as Fetch, False-Variable, and While. This style of translation is described as control-driven meaning that the control tree is responsible for initiating all datapath operations. This approach is robust and flexible but there is a significant drawback: the overhead of the control. The control is nearly always slower than the data and as control and data are frequently synchronised, the data is frequently stalled waiting for the control to catch up.

An example will now be given that attempts to demonstrate how the control-driven structure contributes to control overhead. Figure 3.1 generalises the structure of a control-driven procedure which produces an output (O) and requires an input (A). Internally the process uses two variables (V0 and V1). The

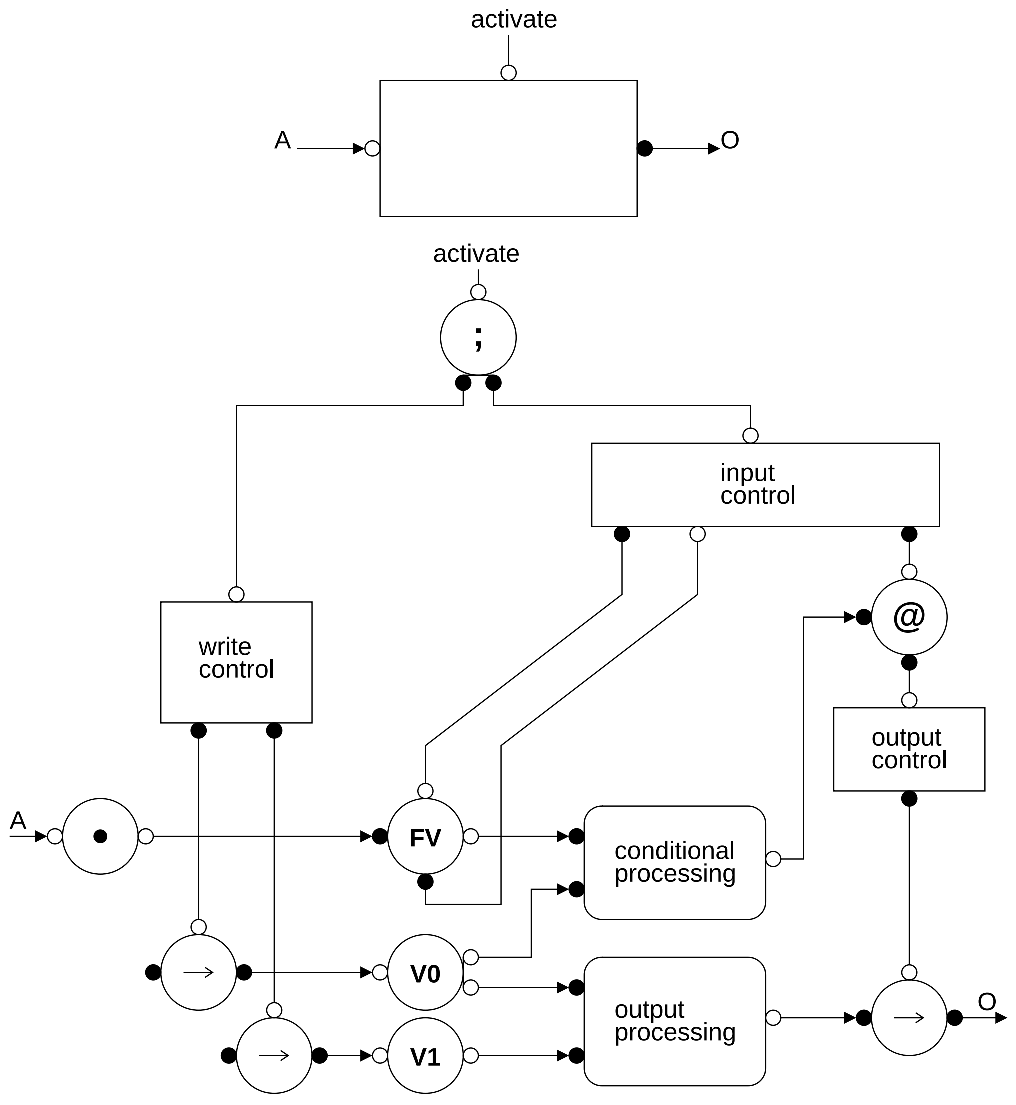


Figure 3.1: General control-driven structure

operation of this structure is extremely sequential. Firstly the portion of control labelled write is activated. The control decides whether to write some data to the Variable components. Once any data is written, it can be considered as being available for reading from the Variable components. However, the control must then complete its handshake before the right-hand side of the tree is activated.

As well as data stored in variables, data from channels may also be used by means of input enclosure. The control (here labelled input) must activate the pulling of any such data to the FV component. The input control waits for the signal from the FV indicating the arrival of the data. Once again, it is not unlikely that this data has been available for some time on channel A which is awaiting synchronisation at the PassivatorPush in order to deliver it.

All the required operands for the data processing operations have now been collected. The control may then initiate data processing operations. It may be necessary to decide what operations should be performed based on some of the data (e.g. a if a case construct is used). Therefore the control may initiate some data processing operations using a Case component for the purpose of making conditional choices. Following this, the final data processing operations that actually produce the outputs are initiated. These outputs are are written to variables or communicated on output channels.

When the variable writes and/or output communications are complete, the data in the Variables and on input channels may be considered as being no longer required. However, all the handshaking in the control for inputs, conditionals and outputs must be completed before the write control is even activated again to begin the process of deciding whether to overwrite the data in the Variables.

Note that for different processing operations, only a subset of the inputs may be required but all the inputs are synchronised with each other and the control before any operations begin. Furthermore, no inputs are released until after all data operations have completed, even though some may not be required after some operations have completed. If data were released sooner then other parts of the circuit will be allowed to proceed sooner as well.

The three principal problems in the structure of the handshake circuit that contribute to the control overhead are:

- All inputs are synchronised with each other before any further operations are begun. Data is available in Variable components before the read control is even activated. After it is activated, the control then synchronises with all channels that are used as inputs before the control begins to decide what operations to perform. If control were operating in parallel with the arrival of data, then data may not be stalled as long while the control decides what to do. The control may even have resolved itself before the data arrives. If there were no need to synchronise all inputs before any operation can proceed then processing, and control that relies on part of the data, can get a head start and operate concurrently with the arrival of the remaining data.
- The sequential activation of the read and write 'halves' of the control tree. This sequencing is needed to ensure the variable is not written and read concurrently. However, the location of the sequencing in the control tree is far away from the Variable leading to sequential operation of the two 'halves' of the control tree. More concurrent operation of the two halves of the tree should increase performance.
- Data processing operations only begin after the control initiates them due to the pull style of operation. If the data processing were to operate in parallel with the control then the overhead of the control should have a reduced impact.

3.1.1 Balsa features to combat control overhead

Several modifications have been made to the existing Balsa system in an effort to reduce the impact of control overhead.

Control re-synthesis

Attempts have been made to apply control re-synthesis to the control of both Tangram [KVL96] and Balsa [CNBE02, CN02]. Control re-synthesis attempts to improve the performance of the control tree by clustering sections of the control tree, determining the overall behaviour, and synthesising a new controller to implement this behaviour using a controller synthesis tool [CKK$^+$97, FNT$^+$99]. By removing the communications between clusters of components, the resulting controller should improve performance over the original control tree.

Control re-synthesis is effective but limited. Improving the speed of the control tree will obviously help reduce control overhead but only so much improvement can be gained. The control still synchronises with data at the same points and so the sequential operation of the control-driven structure is still maintained. Control re-synthesis is complementary to other approaches to improving control overhead including the data-driven style introduced in this thesis.

In addition to re-synthesis it has been suggested [CN02] that the direction of the data processing in Balsa may be reversed from a pull to a push implementation to enable concurrent operation of control and data processing. However, this assumes that a Variable component will be written and read in every 'cycle' of operation. This is not usually the case in conventional Balsa as the control may only conditionally activate the read or write. The data-driven style does not offer Balsa-style variables in order to avoid this problem; data-driven variables are read and written in every cycle and therefore support a push data processing structure.

Concurrent sequencer

This technique specifically addresses the second of the three issues identified above by applying concurrent sequencing [PN98]. The original Balsa Sequence component has been enhanced to include concurrent sequence behaviour. The concurrent sequencer allows some overlapping between the write and read halves of the control. The read half of the control may be activated at the start of the return-to-zero phase of the write half, instead of waiting for the entire handshake to complete. This allows the write RTZ phase to operate concurrently with the read control. Unfortunately this cannot be done if the control is reversed so that reads precede writes as a write-after-read (WAR) hazard would be inserted [NUK+94]. In some situations a write-after-write (WAW) hazard may also prevent the use of concurrent sequencing. The Balsa compiler has been modified to automatically insert concurrent sequencing where it is safe to use it [PTE05]. Concurrent sequencing provides performance improvements but it is limited in where it can be safely applied and only allows partial overlapping of the sequenced operations.

‘Eager’ inputs

Eager inputs use modified FalseVariable components that activate the control without waiting for the data to arrive. The control is able to proceed up until the point where synchronisation with the data is required and there it stalls until the data does arrive [PTE05]. Since the original publication, further work has shown that there are more conditions than originally thought where eager inputs may not be used, although these conditions do not occur in the nanoSpa processor used as an example design in chapter 5. There is the possibility of automatically detecting when it is safe to use eager inputs, or allowing the designer to decide where they should be used by modifications to the source language, or a mixture of both these options. At this time, neither of these options have been fully integrated into the design flow so eager inputs have not been used in the Balsa nanoSpa used in chapter 5.

While eager inputs allow control to get a head start before the arrival of data, it is still necessary to synchronise all the data and control before releasing the data. Data-driven style inputs allow early control activation without suffering from having to synchronise before releasing the data either.

Source description style

This is technically not a particular feature in the synthesis system but is an important factor in the performance of conventional Balsa designs. The transparent compilation from language to handshake circuit structures gives the designer flexibility at the language level to optimise the resulting circuit. The nanoSpa processor which will be used as an example in chapter 5 has been specifically designed to try and achieve the best possible performance from conventional Balsa. The techniques used in the source description are interesting. The use of variables, in most cases, is restricted to the pipeline registers of the processor. The pipeline registers are implemented using the simple one-place buffer circuit shown in figure 2.9 on page 41. This is a very small handshake circuit with the Sequencer located as close as possible to the Variable and, when combined with the concurrent sequencing, the performance of the pipeline register compares favourably to highly optimised controllers [Liu97].

The logic within the pipeline stages is split into small modules that operate concurrently. Each module, therefore, has a relatively small control tree which helps to reduce control overhead. The control tree of each module has a Loop component at the head and so operates independently from other modules. Each module waits for data on its inputs, processes it and produces outputs. Instead of using a large monolithic control tree to direct the movement of data, small steering and merging modules are used to direct the flow of data. Apart from when modules must synchronise on channels to exchange data, they operate in parallel with one another. Essentially, within the constraints of

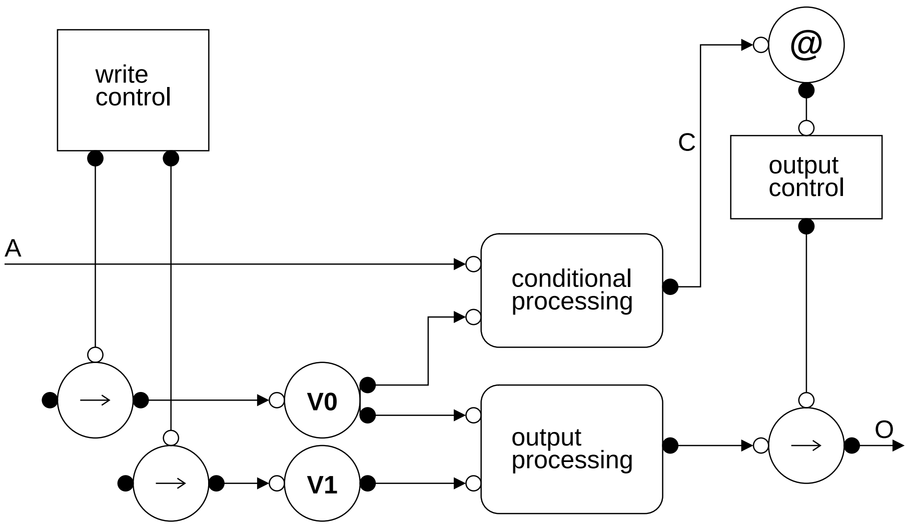


Figure 3.2: General data-driven structure

the control-driven system, an experienced Balsa developer attempts to reduce control overhead by describing a system that is data-driven [PET+07].

At least for an experienced designer, the data-driven style introduced in this thesis is probably more suitable for describing what is desired than the control-driven style.

3.1.2 Data-driven style design aims

The data-driven style has been designed to reduce the impact of all three of the problems identified above. This is achieved by three particular facets of the design style:

- Data-driven control activation. Control is all 'activated' in parallel, synchronising with data only when it is absolutely necessary and releasing it as soon as it has been used.
- Localised sequencing. Sequencing is located local to the variable component. The read and write sections of control can thereby operate entirely in parallel as the localised sequencing ensures that the variable is not concurrently read and written.

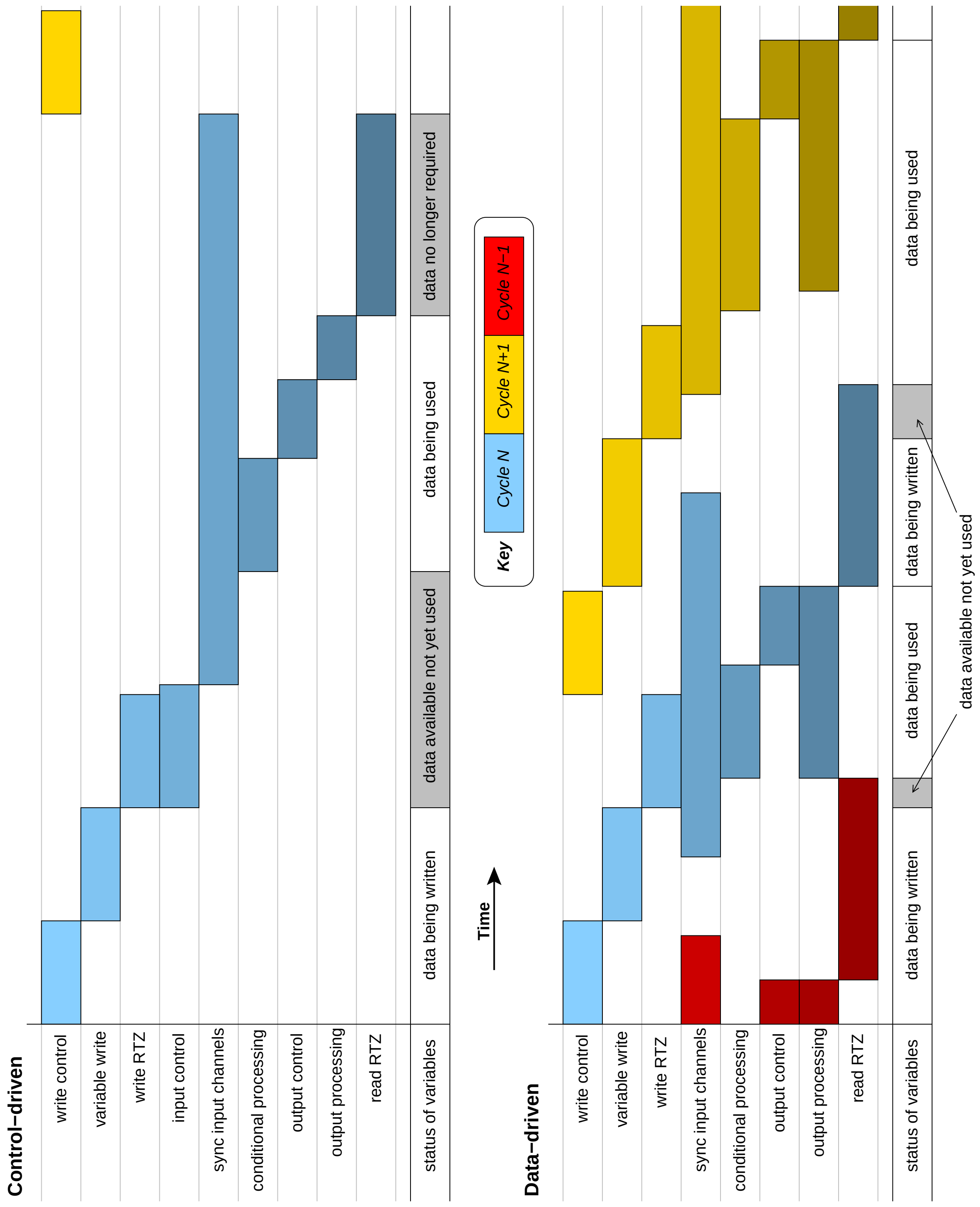


Figure 3.3: Control-driven vs. data-driven

- Speculation. Data processing operations are speculatively executed so they may operate in parallel with the control.

The data-driven equivalent to the structure in figure 3.1 might be pictured as in figure 3.2. Figure 3.3 attempts to give a very rough example of how the data-driven structure enables much greater concurrent operation than the control-driven one. Note how consecutive cycles of operation are overlapped due to localised sequencing and how speculation enables processing to begin earlier as it does not need to wait for the control to activate it. Note also that the periods where data is stored in variables but no use is being made of it are much shorter. This figure is not based on real timings or drawn to an accurate scale and is probably too optimistic but it shows, in general, how the data-driven style aims to reduce the impact of control overhead.

The next two sections summarise the major reasons for the adoption of the proposed approach. Section 3.4 summarises the drawbacks of the proposed approach. Following these sections the data-driven circuit style is introduced.

3.2 Why a data-driven approach?

- A data-driven approach is more commonly used in asynchronous circuit design styles. There are several examples of high performance data-driven style circuits such as the AMULET microprocessors [FGT+97] (also see section 5.1) which were based on the Micropipeline paradigm and the Caltech MIPS [MLM+97] which used the CAST synthesis system (see section 1.1.2).

- The data-driven approach should suffer from less control overhead than the control-driven style of Balsa for the reasons outlined in the preceding section. More parallelism is exploited between data and control by a data-driven style as there is less synchronisation between control and data.

3.3 Why a handshake circuit style approach?

- The handshake circuit approach is not specific to any particular implementation style. A wide variety of possible back-end implementations are possible. The compilation does not map direct to transistors or use unusual circuits such as PCHB so it is much more flexible than some synthesis approaches. Standard-cell implementations are possible.
- Transparent compilation allows the designer to modify properties of the final circuit at the source level. This direct synthesis approach is relatively straightforward to understand. Any valid language description can be compiled into an implementation and there are no complex restrictions placed on the designer.
- The new data-driven style fits into an existing, proven design flow. This saved time and effort in the development of the style as existing tools and components are re-used. It allows integration of the control-driven and data-driven styles allowing the designer to select an appropriate style for different parts of a design.

3.4 Why not a data-driven approach?

- Due to the more restrictive data-driven style, data-driven descriptions are less flexible than those of conventional Balsa. In particular, the nature of Balsa variables means they can be used in a fairly standard fashion familiar to most programmers but data-driven variables cannot. Additionally, no conditional iterative control structure is available in the data-driven style although these are less frequently used. This reduction in flexibility is counter-balanced by the considerable gains in performance achievable if these features are removed.

- Circuits in the data-driven style are likely to require more area and to consume more energy. The localised control of the data-driven style consumes more area than the control-driven tree as instead of appearing once, the control is distributed in many places. This effect is exaggerated in delay-insensitive implementations where an increased amount of completion detection is required and the implementation of push-style variables is particularly expensive. However, the increased concurrency in this distributed control is a major factor in the increased performance. Energy consumption due to switching can also be expected to increase as a result of the increase in concurrent activity. Speculation can also be expected to have an impact on energy consumption as this involves extra switching activity in the datapath that need not occur in the control-driven style.

3.5 Data-driven circuit structures

The data-driven circuit style will be introduced in this section by comparison with the conventional Balsa handshake circuit style. The data-driven style was largely developed by examining and adapting Balsa handshake circuit structures so comparison provides the most instructive method of introduction. General knowledge of the function of handshake components in the Balsa component set is assumed by the following descriptions. Background on the function of many of these components and references to more detailed descriptions can be found in section 2.5. Some new handshake components are introduced and brief descriptions of their operation are given at the appropriate point. Full details of these new components can be found in section 4.2 and appendix B.

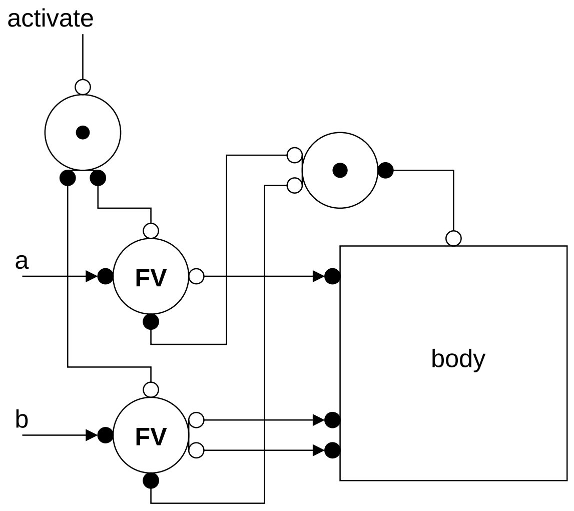


Figure 3.4: Balsa input structure

3.5.1 Input

The conventional Balsa input structure is shown in figure 3.4. This structure is produced by the active enclosure construct shown below.

```
a, b -> then
    <body - a used once, b used twice>
end
```

The activation of the input command, is used to initiate pulling data from the environment on the input channels, (a and b). The FalseVariable (FV) component is used to implement multicast on the input channels. The body of the structure is activated following the the signal ports of the FalseVariable component being synchronised at the Sync component. This activation indicates the availability of the data for the body to then pull it from the read ports of the FV when required.

The data-driven style makes use only of push structures. Instead of using the FV to implement multicast, an alternative push structure must be used. As the input channels are now push channels, there is no need to pull the input data. For inputs that are used in only one place, the data can be pushed

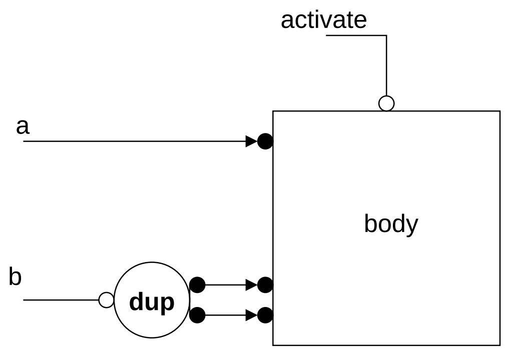


Figure 3.5: Data-driven input structure

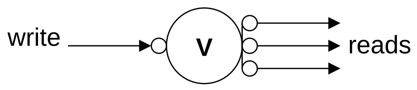


Figure 3.6: Variable component with three read ports

directly to the body. For inputs that are used more than once, a duplicate of the data must be sent to all the required places. The *Duplicate* (*Dup*) component is used to implement this broadcast behaviour. See figure 3.5 for the data-driven version of the circuit example shown in figure 3.4.

An advantage of this approach is that the input channels do not need to be synchronised before activating the body as the body no longer needs an activation to indicate the availability of the data; the data will be pushed to the required places at some point.

The obvious drawback with this approach is that, as the original structure implemented multicast, the body was free to select which read ports, if any, of the FV to use. Where conditional structures are used, the data is only conditionally required. In the broadcast structure, the data is sent to all possible destinations whether they need it or not. The resolution of this problem is discussed in section 3.5.4.

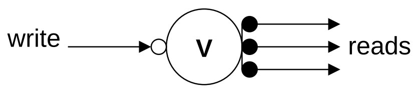


Figure 3.7: VariablePush component with three read ports

3.5.2 Variables

Variables provide data storage within the Balsa language. They are implemented by the Variable handshake component (figure 3.6). This component has a passive input known as the write port and one or more passive outputs known as the read ports. This component allows variables to be very flexible. The control-driven approach allows data to be written to the Variable component by pushing to the write port and read from the variable by pulling from the read ports. The language ensures that the variable is not written at the same time it is read. To the designer, a Balsa variable therefore looks very much like a variable found in most imperative programming languages.

In the data-driven style pull structures are not used so this type of variable is not available. The replacement storage component is called the *VariablePush* and has active push 'read' ports (figure 3.7). Unlike the original Variable component, this component has a write-once, read-once behaviour; each time a data value is written it is automatically pushed on all read ports and the handshake on all read ports must then complete before the next write data is accepted. Instead of a conventional variable, this makes a data-driven variable much more akin to a channel that has storage, thereby allowing each end of the 'channel' to complete independently. This restricted behaviour is a major factor in the somewhat reduced flexibility of the data-driven descriptions over conventional Balsa.

In common with the input structure from the previous section, the drawback of this approach is that the data that is pushed on the read ports of the variable may not actually be required by the destination. If conditional structures are used then the data being pushed on any given read port may not be

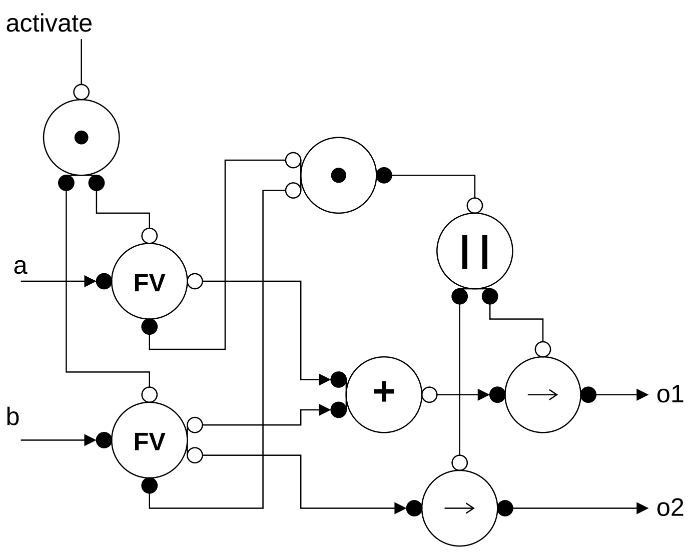


Figure 3.8: Balsa data processing structure

required. The resolution of this problem is discussed in section 3.5.4 which describes the implementation of conditional structures.

3.5.3 Data processing

The original Balsa data-processing structure is a pull structure implemented using the Fetch component to initiate a read of the required data from the required Variable or FalseVariable components, pull it through pass-through data components, and then push it to the destination. The following Balsa code produces the example handshake circuit structure shown in figure 3.8.

```
a, b -> then
    o1 <- a + b ||
    o2 <- b
end
```

This code sends the sum of a and b to the destination channel o1 and sends b to channel o2.

As shown in the preceding sections, in conventional Balsa, Variables and FalseVariables had passive read ports whereas in the data-driven style data

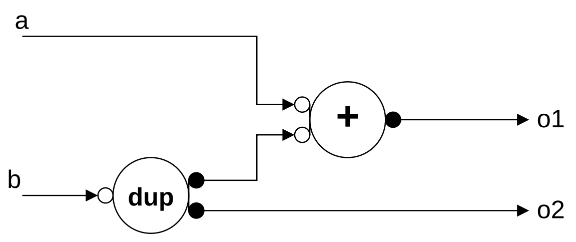


Figure 3.9: Data-driven data processing structure

is always pushed to all places where it may be required. In the data-driven style this data is pushed straight through the push datapath components to the destination as shown in figure 3.9. In this example, both operations are unconditional so there is no need for any synchronisation with control at all and the results are pushed directly to their destinations.

The handshake circuit graph for the data-driven circuit is certainly a lot smaller but what impact does it have on the control part of the circuit? Figures 3.10 and 3.11 show the control for the standard bundled data implementation and 3.12 and 3.13[1] show the dual-rail implementations. A detailed analysis of these circuits is beyond the scope of this discussion but it is clear in both cases, and particularly the dual-rail, that the data-driven circuits are both smaller and faster. Note how in the dual-rail example, the **and** gates are opened early (quite probably before the arrival of data) allowing the data to proceed directly through the datapath logic (the adder in this case). No synchronisation is required between the inputs before they can be processed through the datapath logic and furthermore, the remaining significant control path dealing with the return-to-zero on the inputs has been substantially reduced.

[1]Note that for simplicity the reset has been omitted from the Dup component in this figure.

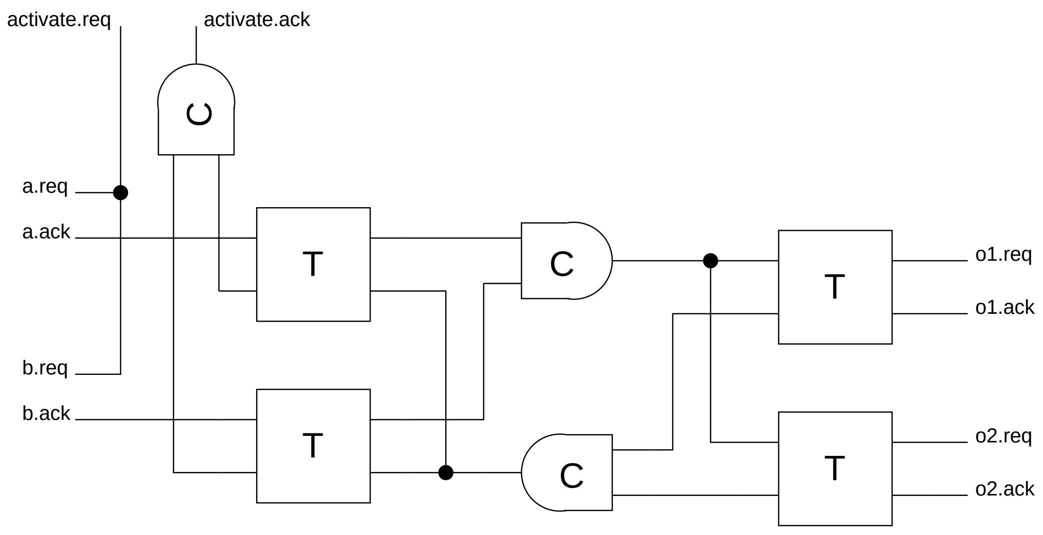


Figure 3.10: Single-rail data processing control circuit

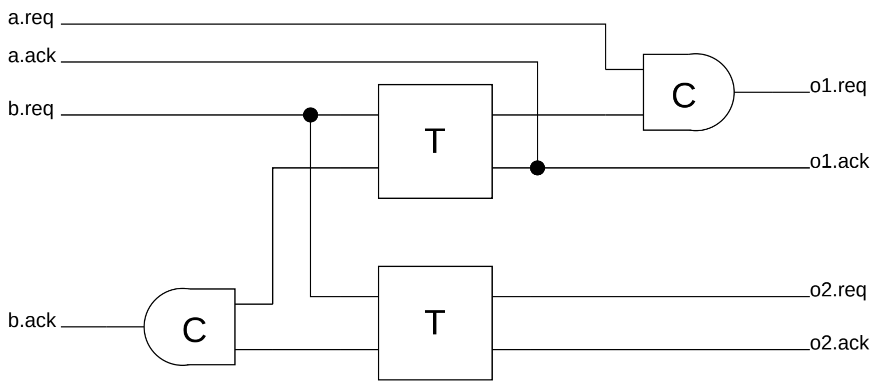


Figure 3.11: Single-rail data-driven processing control circuit

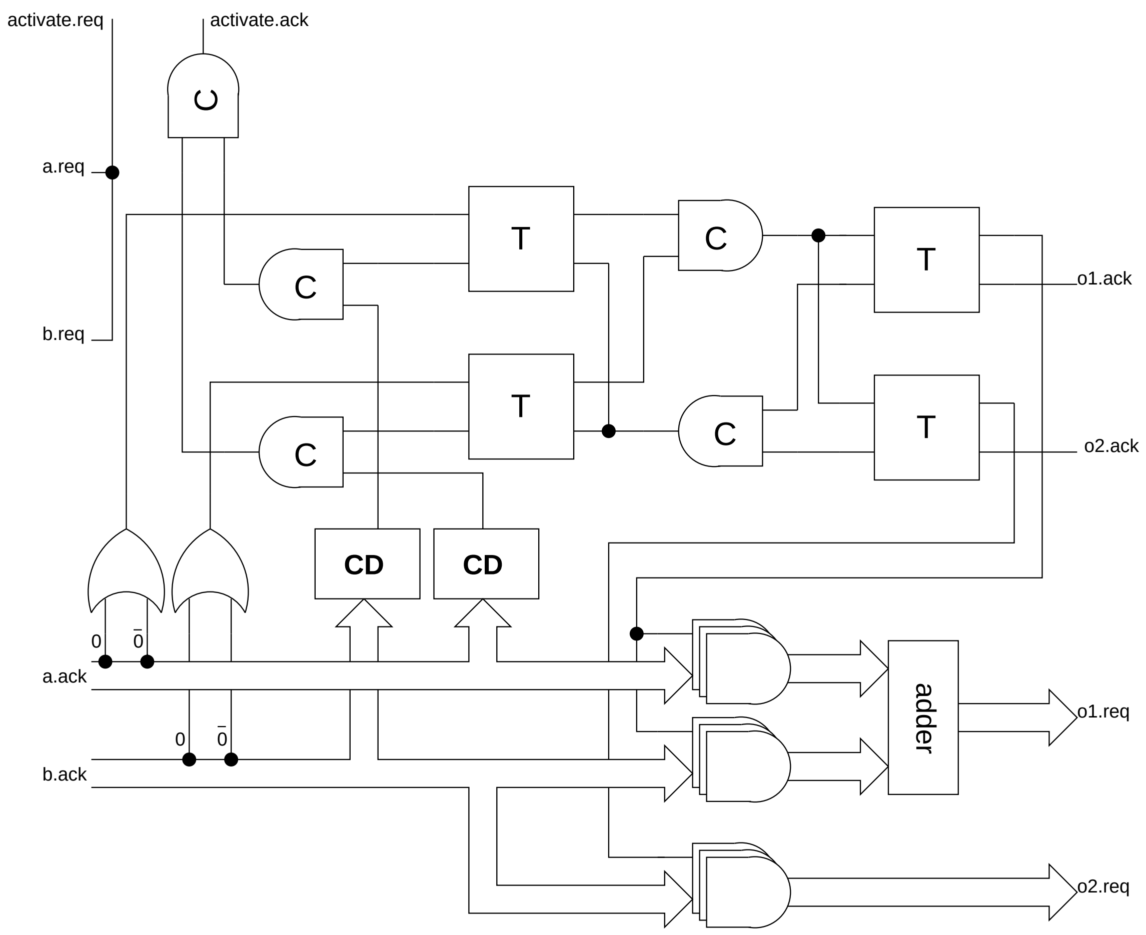


Figure 3.12: Dual-rail data processing circuit

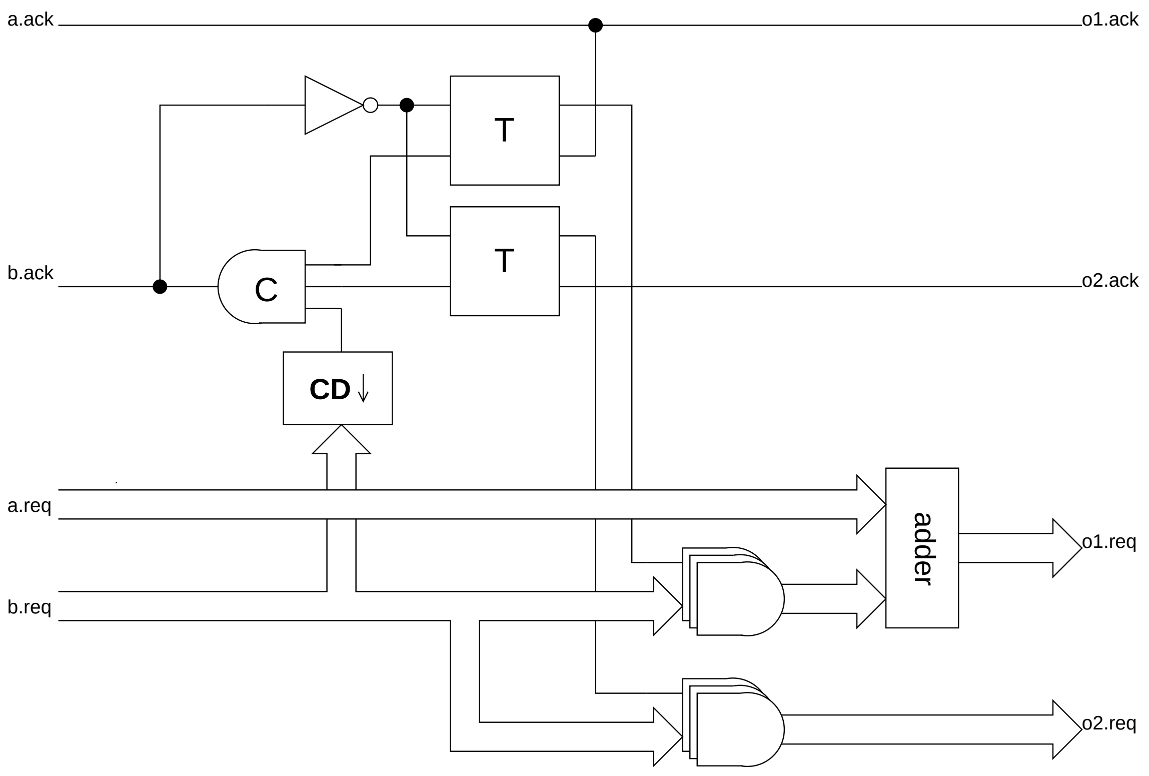


Figure 3.13: Dual-rail data-driven processing circuit

3.5.4 Conditionals

Conditional execution is supported by the `case` and `if` structures in Balsa. This section will take the case construct as an example as it is more commonly used than if, and the implementation of if is fundamentally the same as that of case with a few extensions.

The following Balsa code is an example of the use of the case construct. The control input c is used to determine whether to send the sum of a and b or just b to the output o1. This code is compiled into the handshake circuit shown in figure 3.14.

```
a, b, c -> then
    case c of
      1 then
        o1 <- a + b
    else
        o1 <- b
    end
end
```

As usual, the handshake circuit operates by requesting the three inputs, synchronising on their arrival and then activating the body. The body pulls c from the FalseVariable into a Case component that decides which of its sync outputs to activate based on the value of the control data that has been input. The standard data-processing structure is then used to pull the required data and send it to the output. Additionally in this example, the CallMux component merges the two possible sources for output o1 onto a single output channel. As the Case component will only activate one of its outputs at any time the CallMux will only receive an input on one input channel at a time, thereby avoiding any hazards.

The data-driven equivalent of this circuit is shown in figure 3.15. The difference between the data-driven style and the pull style is that as all inputs are pushed (see sections 3.5.1 and 3.5.2), all the data processing operations are initiated, even though the result may not be required. In order for the circuit

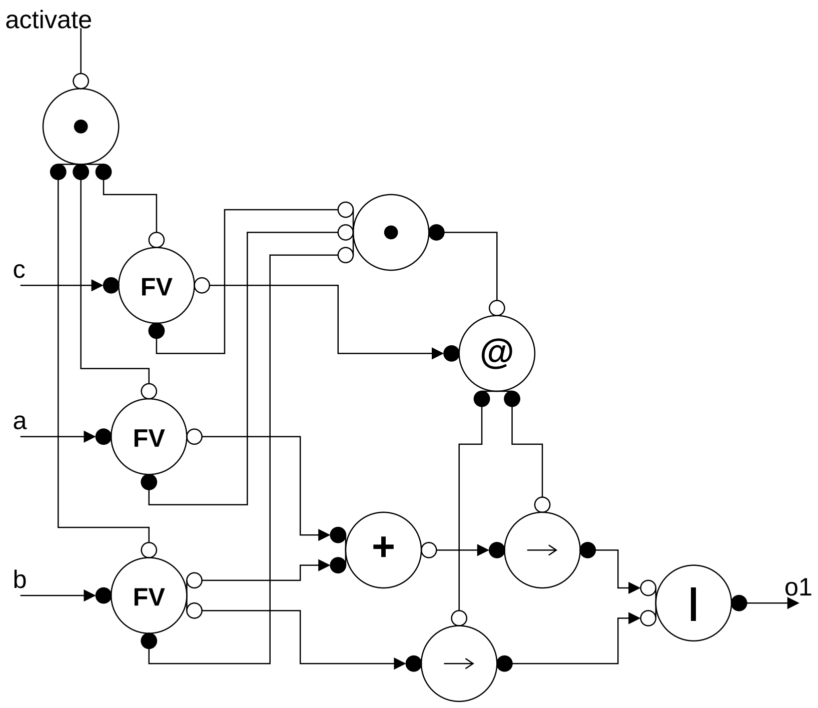


Figure 3.14: Balsa conditional structure

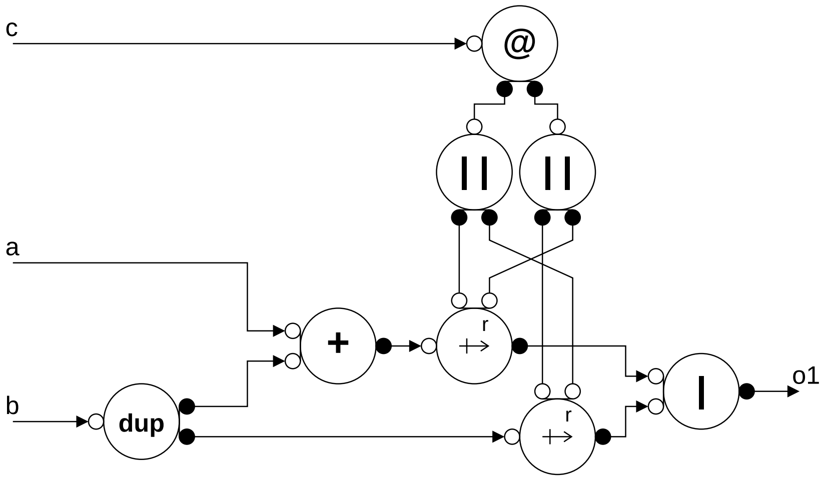


Figure 3.15: Data-driven conditional structure

to operate correctly these extra results must not be allowed to propagate. The *FetchReject* component is introduced to 'reject' the unwanted data. FetchReject is so named because it is rather like a push version of the Fetch component. Instead of pulling data and sending it to the output, it waits for pushed data to arrive on the input and then either passes it through to the output or completes on the input channel without sending anything on the output, thereby 'rejecting' the data. Two sync ports are provided on the component, the activation port which is used to instruct that the data should be passed and the reject port which is used to instruct that the data should be rejected.

Once the FetchReject components are in place, all that remains is to connect the activation and reject ports to the correct outputs of the Case component. In this simple example, one is activated while the other is rejected. This arrangement allows the CallMux component to be used as in the original Balsa circuit because concurrent input handshakes are avoided by correctly using the FetchReject components.

As the data-driven style does not require synchronisation of the inputs, there is potential for performance improvements over the control-driven circuit. The logic in the Case component is able to proceed as soon as the control data arrives, and in parallel with the data processing rather than always having to complete before initiating the pull data processing. However, the data-driven style is essentially speculating on needing the results of all operations. When using a conditional structure the unwanted results must be rejected and the overhead of this operation may harm performance. However, it is believed that generally this overhead should rarely be significant for the following reasons.

As the rejection will often occur in parallel with other useful operations, its effect on the overall performance should be limited. Only where the reject takes longer than useful processing will it reduce the overall performance as both must be completed before the next 'cycle' of the operation. The reject operation itself is quite efficient but if the arrival of the data is slow then the overall impact may be greater.

In cases where there is no operation in parallel with the reject, it may often be the case that the data will arrive in advance of the reject signal and the rejection will therefore be concluded quite swiftly. Note that in the Balsa circuit, it is still necessary for all the inputs to arrive before the operation can complete even if no data processing is actually performed. Furthermore, in the Balsa circuit, the logic in the Case component does not begin evaluating until all the inputs have arrived whereas in the data-driven approach the evaluation can occur in parallel with the arrival of the inputs and so the FetchReject may have received the reject by the time the data arrives so it will at least be immediately rejected, albeit following a possible additional delay through some data processing logic. In the conventional Balsa case, all the inputs must arrive before the process of deciding what to do with them can begin.

Even so, it may be the case that unbalanced datapaths could cause a problem. Consider the example shown in figure 3.15. Here one of the operations is an addition while the other is simply passing through the data from input b unchanged. The addition is most likely to incur a significantly longer delay than the pass-through operation. If the second operation is selected frequently, and assuming the environment can supply inputs and consume outputs quickly enough, there is the potential for the rejection of the add operation to reduce the throughput of the overall circuit.

However, experience in designing with Balsa has shown that the delay of the control nearly always exceeds that of the datapath so it is reasonable to be optimistic that many datapath delays incurred as a result of speculation will be entirely masked by the delay of the control that works out whether or not to reject. Additionally, the inputs needed for the datapath operation may arrive earlier than those for the control allowing the datapath to complete before the control signals arrive at the FetchRejects.

Finally, it is always possible to communicate to the experienced designer information to assist them in avoiding generating situations that may degrade performance. Section 4.3 presents an example of source level optimisation to avoid speculative operations.

3.5.5 Conditional input

Conditional inputs may occur in Balsa code when an input is made as part of the body of a conditional structure. For example, in the code below, channel b is a conditional input in the else clause of the case construct.

```
a, c -> then
    case c of
      1 then
        o1 <- a
    else
        b -> o1
    end
end
```

During the operation of this code, data is only pulled on input channel b if the else clause is activated. Otherwise no communication occurs on channel b. This code is compiled into the handshake circuit shown in figure 3.16.

The important thing to notice when this circuit is converted to the data-driven style is that when data arrives on input b, it is always used; there is no need to reject any data if the else clause of the case is not executed, as the input never arrives. Of course, in a data driven style there may be a request pending on channel b but this should be acknowledged by a subsequent cycle of the circuit when the else clause is executed. It is important that, until the else clause is taken, this request is not propagated too far as a conflict may be caused. To avoid this possibility the *FetchPush* component is used. This component can be considered as a push version of Fetch, or a version of FetchReject without a reject.

To further explain the above, consider the example in figure 3.17 which

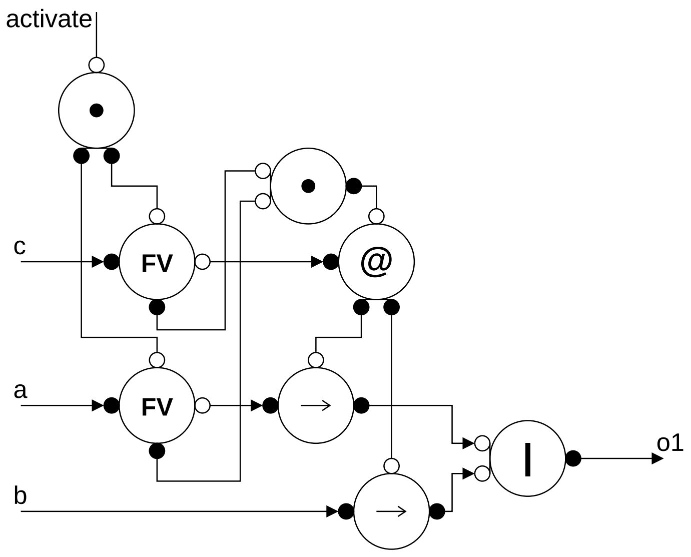


Figure 3.16: Balsa conditional input structure

is the data-driven equivalent of the example in figure 3.16. The FetchPush component is used on channel b to ensure any request on b is not passed to the CallMux component before the Case has decided that operation should occur. This ensures the inputs to the CallMux cannot occur concurrently.

Due to the pull nature of conventional Balsa handshake circuits and the use of the FalseVariable component, following the arrival of inputs (whether conditional or unconditional), these inputs can then be read and combined in any desired fashion. In the data-driven style this flexibility is not so readily available. Consider the following code example, only a small modification to the last example given above.

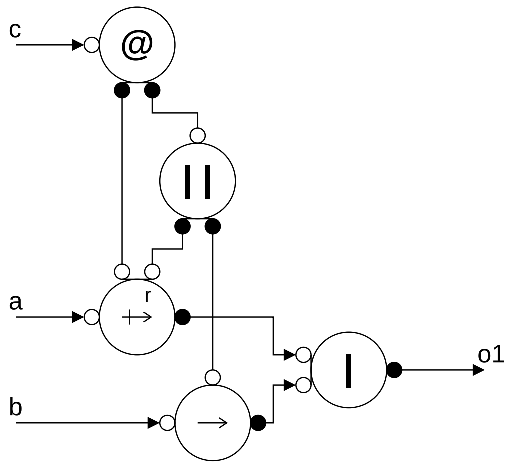


Figure 3.17: Data-driven conditional input structure

```
b, c -> then
    case c of
      1 then
        o1 <- b
    else
        a - > then
            o1 <- a + b
        end
    end
end
```

This code presents no problem for the Balsa compiler but a data-driven equivalent is more difficult to derive. If the approach given to this point is followed for this example then input b will be duplicated and sent on a direct path to o1 and through the adder with a and then to o1. If the else clause is executed then there is no problem. However, if the else clause is not executed then what should be done? Where input a was also unconditional, a reject was used to kill the unwanted data after the addition had occurred (see figure 3.15). In this example, if the else is not taken then there will be no data on a to reject, or more accurately, any data that is pending on a is not to be rejected. In general, this problem will occur any time conditional inputs are combined in an expression with unconditional inputs.

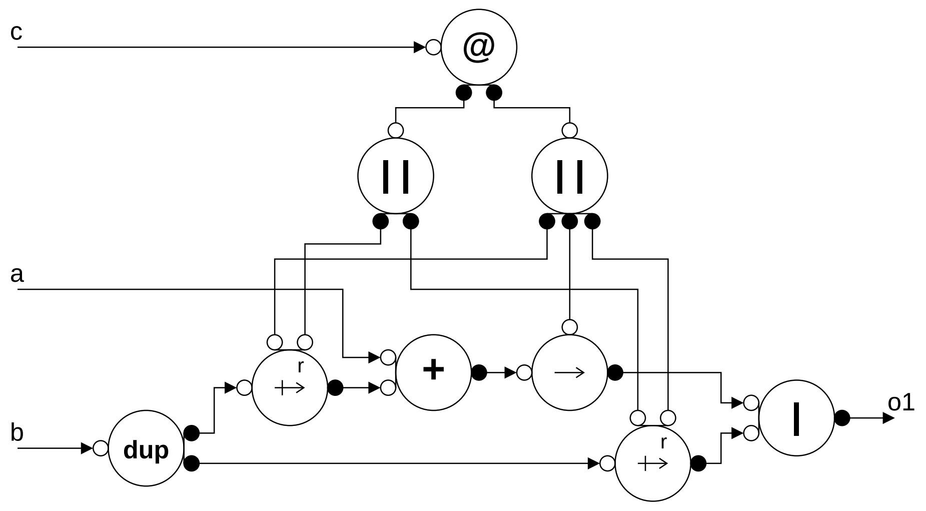


Figure 3.18: Combining conditional and unconditional inputs

In order to avoid this problem, a complex scheme could be devised to reject the unconditional inputs (if they are not required) before they are combined with the conditional inputs. For example, a circuit similar to the one in figure 3.18 could be used. However, such a scheme reverses part of the advantage of adopting a push style as the datapath operations are once again stalled waiting for control to decide whether the result of the operation is required, instead of control and datapath operating in parallel. Furthermore this scheme presents additional complexity in compilation as the placing of rejections is now much less straightforward.

For these reasons, such a scheme has not been used. Instead, combinations of conditional and unconditional inputs within expressions are considered invalid by the compiler, avoiding the need to produce an implementation at the expense of some reduction in flexibility. However, the user is still able to implement this scheme in the source description if they choose. See section 3.6.8 for an example of this.

3.5.6 Nested conditionals

Conditional structures in Balsa can be nested within one another as demonstrated by the following code.

```
c, d -> then
    case c of
      1 then
        case d of
          1 then
            <body X>
        end
    else
        e -> then
            case e of
              1 then
                <body Y>
            end
        end
    end
end
```

In the control-driven style the output activations from one conditional structure are simply used to activate the nested conditional. In the data-driven style, the evaluation of the logic in all Case components proceeds concurrently, but the output activations of nested conditionals must be delayed pending an activation from the outer structure as shown in figure 3.19. This example demonstrates the use of the *CasePush* and *CasePushR* components.

CasePush is used where it is necessary to synchronise with an activation before output activations are made from the Case component. This is the case for the Case component whose input is channel e as data will only arrive on e when it is required.

It may be necessary to reject the input to a CasePush if data will arrive that is not required, as in this example with the Case component whose input is channel d. CasePushR is simply a CasePush with a reject input that upon activation will discard the input data without activating any outputs. The reject

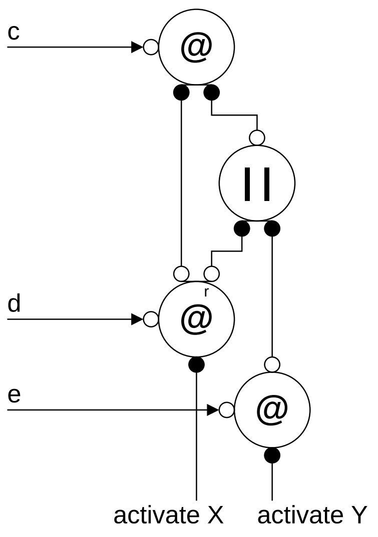


Figure 3.19: Data-driven nested conditional structure

port is then activated on all conditions where the activate port is not.

3.5.7 Arbitration

The nature of asynchronous design means that it is sometimes necessary to use arbitration to determine the order of arrival of independent inputs. The arbitrate construct is used in Balsa to generate a circuit that will arbitrate between two sets of inputs. The following code gives an example of its use.

```
arbitrate a then
    o <- a
| b then
    o <- b
end
```

In this code, the first section of the arbitrate is activated if input a arrives and the second section if input b arrives. If both inputs arrive concurrently then it will arbitrarily select one as having arrived first and activate the appropriate section. The Balsa implementation of this code is already partly data-driven as the inputs are push rather than pull.

This code is compiled into the handshake circuit in figure 3.20. As usual,

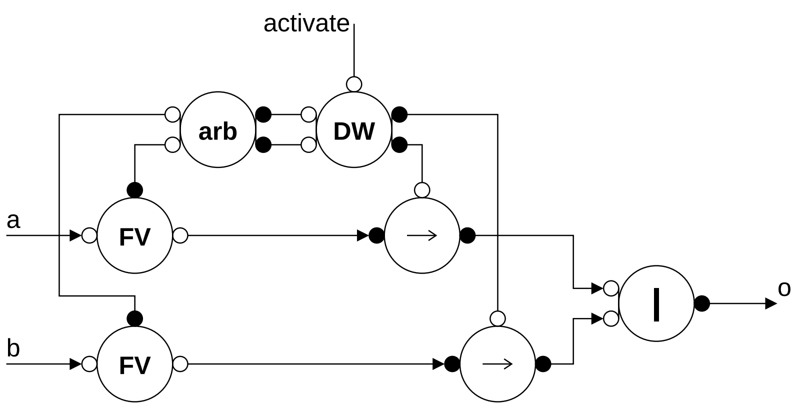


Figure 3.20: Balsa arbitration structure

the FalseVariable components are used to receive inputs but in this case the inputs are push rather than pull so a FalseVariable with a passive input is used. The signals from the FalseVariables are used as the inputs to the Arbitrate (arb) component. The Arbitrate component passes through handshakes on its passive ports to the corresponding active port but ensures that output handshakes are mutually exclusive. Should both inputs arrive concurrently then a non-deterministic decision will be made as to which to pass first. The outputs of the Arbitrate are used to feed a DecisionWait (DW) component which synchronises with the activation from the control tree before activating the appropriate output.

The data-driven equivalent shown in figure 3.21 is very similar to the original Balsa handshake circuit. The only difference is that the FalseVariables have active push outputs and conversely the Fetch components must be replaced by FetchPush with a passive input.

This structure is a variety of conditional structure so inputs that are not part of the arbitration but are used in the body of the arbitrate need to be rejected from the DecisionWait. However, note that all inputs to an arbitrate are used, even if the other side of the arbitrate is activated first, so no reject is necessary for these. Therefore, the inputs being arbitrated are conditional and the same restriction on combining them with unconditional inputs as described in

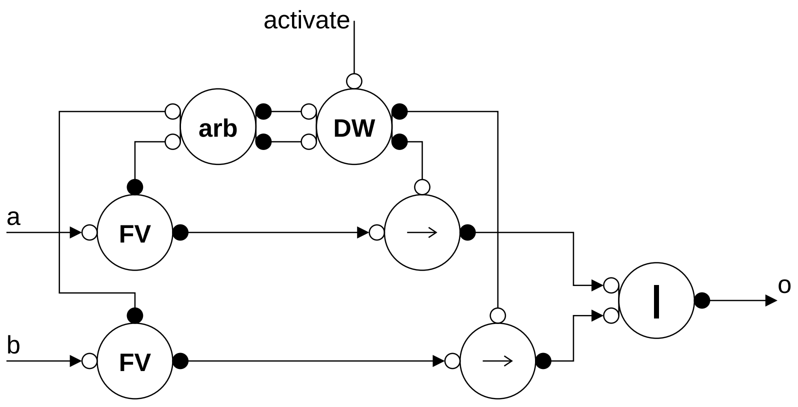


Figure 3.21: Data-driven arbitration structure

section 3.5.5 applies here also.

3.5.8 Arrayed variables

In Balsa, variables that are declared as having an array type may be implemented in two principal ways. The first is used when the entire array is written at the same time and elements are not written individually. For example, consider the code fragment below.

```
input  i : array 0..3 of 2 bits
input  c : 3 bits
output o : 2 bits

variable v : array 0..3 of 2 bits

loop
    i -> v ;
    c -> then
        case c of
          0b1xx then
            o <- v[(#c[0..1] as 2 bits)]
        | 0b0xx then
            o <- v[0]
        end
    end
end
```

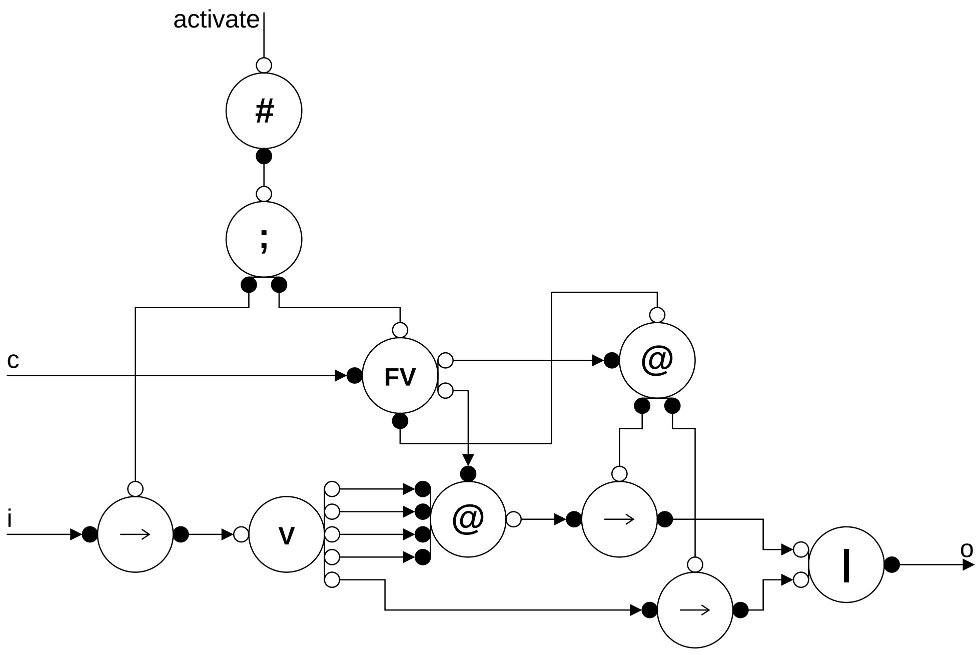


Figure 3.22: Balsa single-write array variable structure

This code is implemented by the handshake circuit in figure 3.22. This code demonstrates writing a single value to the entire array and then reading individual elements. The code also shows how elements may be specified with a constant index or by a non-constant run-time index. (Of course, the entire array may also be read in one go.)

In this instance a single Variable component is used to implement the variable. To support reading the variable with a constant index, a read port is placed that provides the correct bits from the array. To support reading the variable with a run-time index, a read port is generated for each element in the array. The CaseFetch component is then used to select the correct element based on the index.

It is relatively easy to derive a data-driven equivalent for this type of arrayed variable structure. Figure 3.23 shows a data-driven equivalent of the circuit in figure 3.22. The new component used in this circuit to implement the run-time index is the *Mux*. This component receives an index and uses this

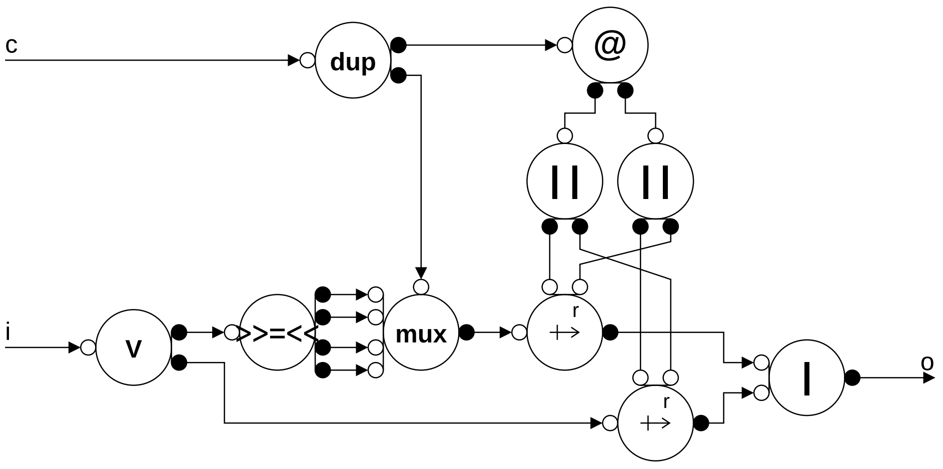


Figure 3.23: Data-driven single variable array structure

to select one of its inputs to pass it to the output, discarding the other inputs. The SplitEqual component is used to generate the individual array elements as adding multiple read ports to the variable would incur a significant overhead that can be avoided in this situation where the destination of all the data is to be the same place.

The second type of variable is that where elements in the array are written individually. This situation introduces significant added complexity as the example code below and circuit in figure 3.24 demonstrate.

```
input  i : array 0..3 of 2 bits
input  c : 3 bits
input  d : 2 bits
output o : 2 bits
output p : array 0..3 of 2 bits

variable v : array 0..3 of 2 bits

i -> v ;
loop
    c -> then
        case c of
          0b1xx then
            o <- v[(#c[0..1] as 2 bits)]
        | 0b0xx then
            d -> v[#c[0..1] as 2 bits)]
        end
    end
    ;
    p <- v
end
```

This code demonstrates the full flexibility offered by Balsa for using arrayed variables. Firstly, a single value is written to the entire array, then an individual element is read or written, and then the entire array is read as a single value.

The strategy adopted by Balsa is to implement the arrayed variable using multiple Variable components, one for each element in the array. The control can then initiate reads and writes of the passive ported Variables individually or as a group, splitting the write data and combining the read data as required.

A data-driven equivalent of this circuit structure presents substantial problems. Once the Variables have been converted to VariablePush components, it is necessary to write to each VariablePush before it is read. After writing to a single element in the array, only that element would be available to read. This behaviour could be adopted by the data-driven approach but it would create difficulties for a compiler. When compiling a read from the array it cannot be

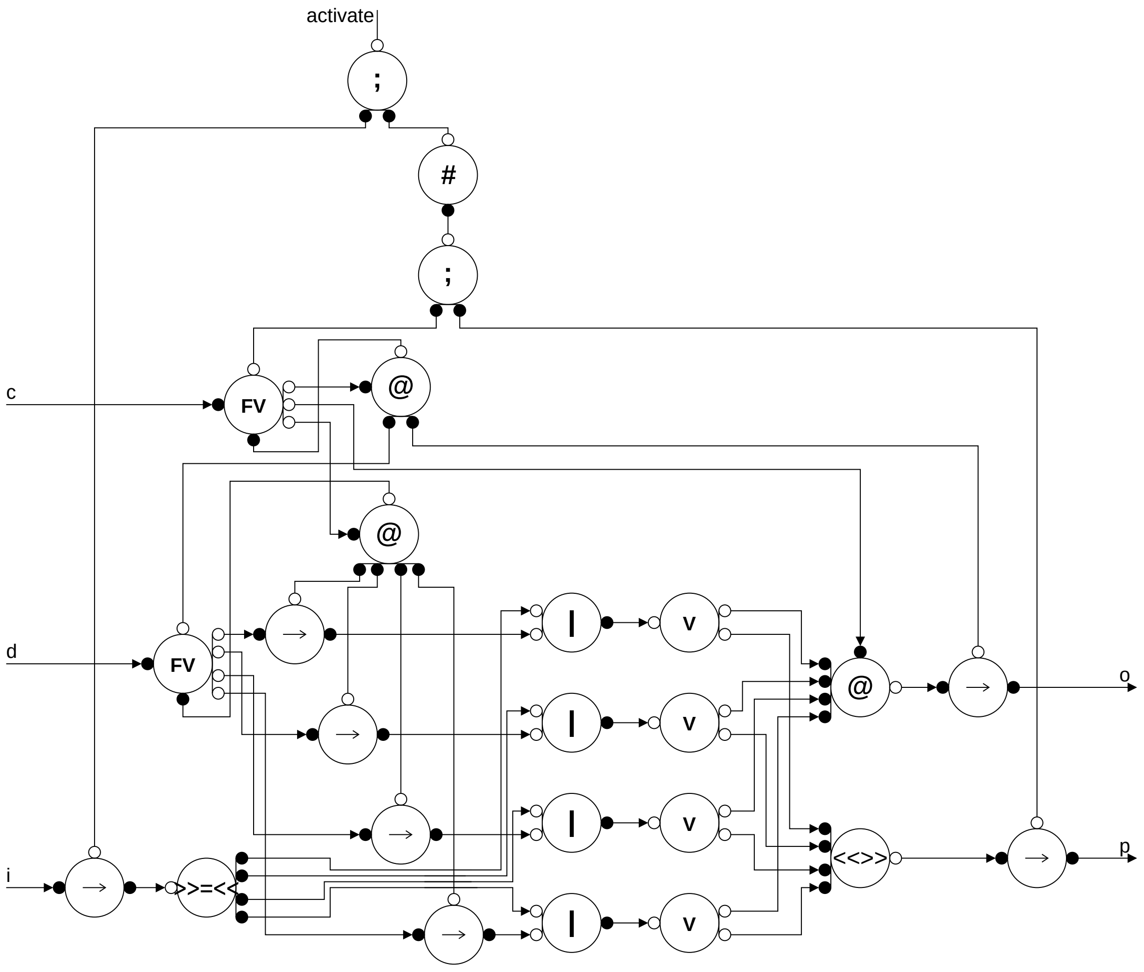


Figure 3.24: Balsa general arrayed variable structure

assumed that all the elements will be being pushed so it is not possible to simply reject the remaining elements. An option is to leave the management of the structure to the user, who must only attempt to read elements of the array that are written. Alternatively the user could be restricted to always writing to every element if they wish to use run-time indexing.

Alternatively, an elaborate scheme to write-back the original data to those variable elements that are not written could be devised. This would ensure that every time any element in the array is written, all the other elements are also written (with unchanged data). To the read side, the arrayed variable always appears as if the entire array has been written, enabling use of the read structure shown earlier (figure 3.23).

Neither of these suggestions have been fully adopted in the data-driven style though the first option has been adopted in part. Instead, two different types of arrayed variable are provided. The first produces a single VariablePush component that must be written in its entirety but allows individual elements to be read as in figure 3.23. The second generates multiple, essentially independent, VariablePush components, very much as arrays of channels are available in Balsa (and the data-driven style). These may only be written using constant indices; run-time indexing cannot be used for writes. Run-time indexing may be used for reads but where it is used it will be assumed that all elements will be available to select from. The user must therefore ensure that all elements are written when using a run-time index.

This second type of variable can be used by the user to generate a fairly close approximation of the functionality of the multi-variable Balsa structure by implementing, in the source description, the second of the schemes offered above. Although the functionality may be similar, the area used is substantially greater. For an example of code that generates this structure, see section 3.6.6 and the register bank of the nanoSpa in section 5.4.3.

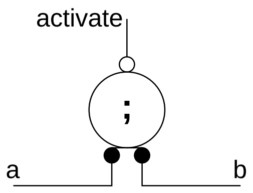


Figure 3.25: Sequenced sync structure

3.5.9 Sync channels

Sync channels are available in the data-driven style and are implemented in the same fashion as in Balsa. It is expected that there will be little use in a data-driven style for data-less channels, but they are notable in that the sync command is the only command in the data-driven style that may be explicitly sequenced. For example, the following code is compiled to precisely the same structure in Balsa and data-driven programs; the structure being a Sequence component (figure 3.25).

```
sync a ;
sync b
```

Sync channels may be used as 'inputs' to the arbitrate structure.

3.6 New input language

In the preceding sections the new data-driven circuit structures were described. This section will describe the high-level language that is translated in a syntax-directed fashion into those circuit structures. The language is designed to resemble conventional Balsa wherever possible. Therefore the description below attempts to highlight the differences and where it remains silent it can be assumed that the Balsa solution has been directly adopted.

Unlike Balsa where a circuit consists of commands linked by sequential or parallel control, the data-driven approach consists of lists of commands that operate independently and in parallel. Unlike the control-driven approach,

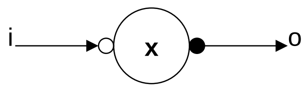


Figure 3.26: Data-driven one place buffer

control sections of the circuit do not wait for an activation but proceed as far as they are able, pausing only when awaiting data.

3.6.1 Hello World!

The equivalent of a Hello World program in Balsa is the one place buffer. This serves equally well here as an initial introduction to the data-driven language.

```
-- One place buffer
procedure buf (input  i : 1 bits;
               output o : 1 bits ) is
    variable x : 1 bits
begin
    input  i
    output x
    during
        x <- i
    end

    input  x
    output o
    during
        o <- x
    end
end
```

It can be seen from this small example that much of the language is very similar to conventional Balsa. The declaration of the procedure and the input and output ports is identical. Unlike conventional Balsa, the procedure input ports will always be passive due to the push style of implementation. Internally to the procedure the input ports are treated as read-only channels and the output ports as write-only channels.

The main new feature in evidence here is the division of the procedure into blocks consisting of input and output declarations and a body containing the commands that use the inputs and generate the outputs. Unlike Balsa, the control structures of the circuit are largely implicit. Blocks implicitly operate in parallel, as do the list of commands within the block. The only synchronisation between the two blocks in this example takes place at the variable; the read must complete before the next write can overwrite the data in the variable. This allows the variable reads and writes to overlap to the largest possible extent.

Incidentally, for comparison with figure 2.9 on page 41, the handshake circuit for this buffer is simply a VariablePush component (see figure 3.26).

3.6.2 Variables

The control-driven style of Balsa allows variables to be accessed in a very general fashion, so as to appear very similar to variables in a standard programming language. Variables can be read and written in any arbitrary sequence. The Variable component has passive read and write ports and the control tree initiates communication on these as required. In the data-driven approach, the VariablePush immediately pushes any data written to it out of its active 'read' ports. This means that a variable must always be read after it has been written. Variables therefore resemble less those of standard programming languages and are much more similar to channels. In fact, it may be more helpful to think of a variable in the data-driven style as a channel that contains storage, or even as a type of channel which each communicant can use at different times, rather than having to synchronise like a normal channel.

Reflecting this, variables are specified as inputs and outputs (to blocks – procedure ports only connect using channels) in precisely the same fashion as channels. In the following discussion use of the term channel generally implies a channel or variable except where otherwise stated.

3.6.3 Input 'control'

As all inputs are passive, it is not necessary to generate requests to pull the inputs as in Balsa. Apart from this, the semantics of the input are similar to the 'eager' inputs described in section 3.1.1 in that the 'control' is activated early. However, in the eager semantics, it was still necessary for the control and data to synchronise to release the data once all required reads had been completed on the channel. As reads are now to be pushed, this synchronisation is unnecessary as the release of all the 'read' ports will indicate that all reads on the channel are completed.

In the data-driven approach, therefore, inputs are merely specified as arriving at some point during the operation of the commands; the control waits for the arrival of inputs at any points where they are read (if they have not arrived already).

3.6.4 Write command

The write command (<-) is used to output the result of an expression to an output channel (or variable). The channels written to must have been declared as an output from the block.

Compilation of the write command involves compiling the expression into appropriate push datapath components and connecting the result to the destination. This may be a direct connection or it may be through a FetchPush or FetchReject depending on whether the command is conditionally executed and whether a rejection is required to discard the result if it is not required.

3.6.5 Arrays

Channels and variables can be arrayed in a similar fashion to Balsa. However there are some differences in the semantics of variable arrays. In Balsa, a variable declared as having an array type will generate a separate variable for each item in the array, but a single read and write structure allowing access to only one item in the array at a time. A similarly declared variable in the data-driven language generates a single variable that holds an entire value of the array type. The whole of the array must therefore be written to at one time.

Variables can also be declared in a similar fashion to arrayed channels producing multiple variables in the implementation. Each of these variables must be written individually; the whole array may not be written by a single command (with the exception of writing the same value to every item in the array using the `all` keyword as described in the next section). Furthermore, if a non-constant index is used to access the array then it is assumed by the compiler that all elements of the array will be available to select the correct element from. Therefore, a write must be made to all elements in the array where a non-constant index is used to read from the array or a deadlock may be shortly anticipated.

In addition to the above usage, this type of array may be used as a set of essentially independent variables accessible by constant indices. This is a particularly useful feature when used with structural iteration as demonstrated in the next section.

3.6.6 Structural iteration

Structural iteration is a very useful language feature especially when combined with arrayed channels and variables. Essentially it allows the same code to be compiled multiple times with different channel and variable connections. For example, the following code is a simplified excerpt from the register bank of the nanoSpa processor (see also section 5.4.3).

```
constant REGNUM = log REGCOUNT bits

array REGCOUNT of variable reg_usrw
array REGCOUNT of variable reg_usrr

input  reg_usrr
output reg_usrw
during
    for i in 0..REGCOUNT - 1
        reg_usrw[i] <- reg_usrr[i]
    end
end


input  reg_usrw, wc, wd
output reg_usrr, reg_svcr
during
    foreach i in reg_usrr
        case wc of
          (i as REGNUM) then
            reg_usrr[i] <- wd
        else
            reg_usrr[i] <- reg_usrw[i]
        end
    end
end
```

The above code generates REGCOUNT instances of the circuit in figure 3.27. (The position of the channels that take data to the read ports are indicated on the diagram but the code for the read ports is not given above.)

Effectively this code generates a register 'cell' for each register. In each 'cycle' of operation the write control (wc) and data (wd) is duplicated to each cell and that cell compares the register address in the control against its own index. If they match then the write back data is written to that register, otherwise the original value from the register is written.

Three forms of structural iteration are supported. The `for` and `foreach` constructs are demonstrated in the above example. `For` allows iteration over a given range and `foreach` allows the range to be specified as the size of a

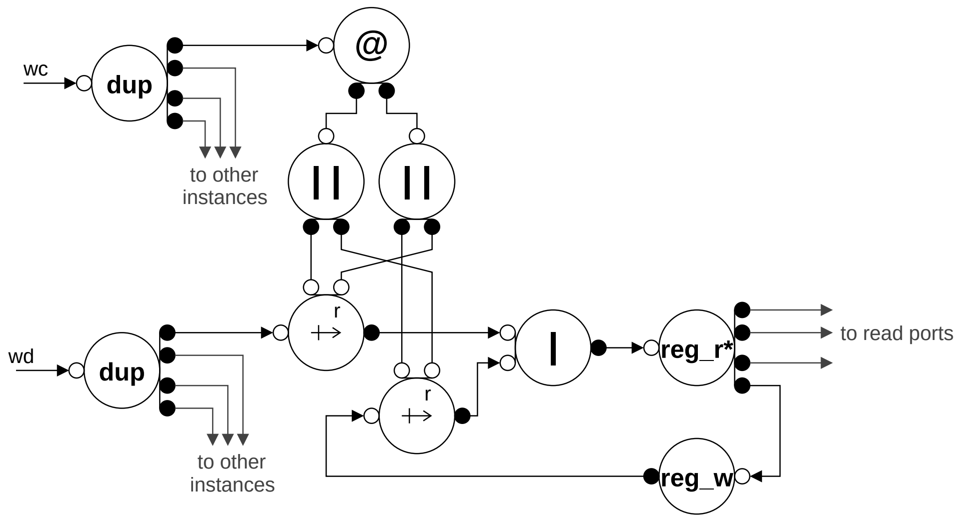


Figure 3.27: Simplified register cell

given arrayed variable/channel.

The final form allows an array to be the target of a write command by prefixing the command with the `all` keyword. This provides shorthand for allowing writing of the same value to an arrayed member. For example the following two code fragments are equivalent functionally.

```
-- This code is functionally the same as ...
foreach i in an_arrayed_output
    an_arrayed_output[i] <- an_expression
end

-- ... this code
all an_arrayed_output <- an_expression
```

The `all` keyword allows for the possibility of a smaller implementation as any datapath components in the expression need only be generated once with the result being passed through a Dup component to send the result to each variable in the array.

3.6.7 Initialisation

A special `init` block may be included in each procedure; its purpose being to initialise variables to a particular value and alter the usual behaviour of the variable so it first pushes this value before accepting a write. The `init` block simply consists of a list of write commands with a variable as the target of the write and an expression that can be evaluated at compile-time. The `all` keyword (see section 3.6.6) may also be used to load all items in an arrayed variable with the same value.

This initialisation should not be confused with the initialising of a variable to a value in Balsa. Variables must only be initialised if it is desired to read from them before writing to them. In Balsa it is acceptable to initialise a variable to a value just in case it is read before the first time it is written, or even if it is always written before the first time it is read (although it is not sensible to do so in the latter case).

3.6.8 Restrictions

Combining inputs

A conditional input is an input that is part of the body of a conditional structure. As explained in section 3.5.5, such inputs cannot be combined with unconditional inputs in any expression. So the following code will produce a compiler error as the operation a + b cannot be used where the input of a is conditional and b is not.

```
input  b, c
output o1
during
    case c of
      1 then
        o1 <- b
    else
        input a during
            o1 <- a + b
        end
    end
end
```

This restriction can be worked around by declaring another channel and making both inputs to the expression conditional as follows:

```
channel t

input  b, c
output o1, t
during
    case c of
      1 then
        o1 <- b
    else
        t <- b
        input a, t during
            o1 <- a + t
        end
    end
end
```

Note that by using this technique, less advantage is taken of the speculation as the case must be resolved before the channel t is written and the expression begins evaluation. Note also however, that the speculative evaluation of the addition is avoided in the case where the else clause is not chosen. This may be exploited for the purposes of improving performance or reducing energy consumption.

All inputs and outputs must be used

All inputs and outputs that are declared must appear in the body of the block. (They must also be declared if they appear.) It is only necessary for the possibility to exist for each output to be produced. It is not necessary for every, or indeed any, output to actually be produced by the block when it is operating. Once an input is declared it will be assumed that a value will arrive from that channel or variable, but declaring an output declaration means only that the block is the one that writes to the channel/variable, not that a value will definitely be written in any particular 'cycle' of the block.

Output to input dependencies must not be disjoint

An output depends on an input if the input must arrive before the output can be produced. For example in the following code o1 depends on c and a, t depends on c and b and o2 depends on a and t.

```
channel t

input  a, b, c, t
output o1, o2, t
during
    case c of
      1 then
        o1 <- a
    else
        t <- b
    end

    o2 <- a + t
end
```

This gives three sets of input dependencies for each output: {c,a}, {c,b}, {a,t}. These are not disjoint as c appears in the first two and a appears in the first and third. This code is therefore valid.

The following code is not valid:

```
channel t

input  a, b, c, t
output o1, o2, t
during
    case c of
      1 then
        o1 <- b
    else
        t <- b
    end

    o2 <- a + t
end
```

The sets of input dependencies for this code are: {c,b}, {c,b} and {a,t}. The set containing a and t is disjoint from the other two sets. A separate block should be used to produce o2:

```
channel t

input  b, c
output o1, t
during
    case c of
      1 then
        o1 <- b
    else
        t <- b
    end
end

input  a, t
output o2
during
    o2 <- a + t
end
```

This rule helps to ensure the design is understandable as each block has a single 'cycle' of operation due to the fact that all inputs are synchronised somewhere, though not necessarily with all others. For example, if the following code were valid then its meaning would be open to question but presumably, following the method of operation so far defined, o1 will be written every time a arrives, o2 would be written every time b arrives and there would be no synchronisation between the two operations.

```
input  a, b
output o1, o2
during
    o1 <- a
    o2 <- b
end
```

In Balsa, if one were to write: `o1 <- a || o2 <- b`, then there is an explicit synchronisation that takes place in the control. The data-driven style is designed to avoid making such synchronisations. In Balsa, there will be one communication on o1 and one on o2 before another takes place on either channel. In the data-driven style there could be infinite communications on o1 before any occur on o2 or vice-versa. This could make designs much more difficult to understand.

3.7 A note on temporal iteration

As described in section 2.5.5, Balsa features a conditional while loop structure that is similar to constructs found in most imperative programming languages. The control-driven style allows this control structure to be implemented easily. The data-driven style is based on data flow rather than explicit control structures and so cannot implement temporal iteration in the same manner. A couple of points are worth noting about the conditional looping structure. Firstly, it is not particularly common – the Balsa nanoSpa design contains only two while loops, both of which are related to the decode of multi-cycle instructions (see section 5.4.2 on page 139). Secondly, it is inefficient (see section 5.4.2 on page 139 and tables 5.1 and 5.3 on pages 152 and 154).

The advantage of the control-driven iterative structure is that it is possible to read from a variable many times while the iterative process is operating. It is, however, perfectly possible to implement such iteration in the data-driven style but it must be described in a data-flow fashion. For example, consider a loop that generates some outputs based on the values of two variables, a and b. This can be implemented by code along the lines of that given below. The variables la and lb are used to 'feed-back' the values of a and b to each iteration of the loop.

```
variable a, b
variable la, lb

input  a, b, ...
output la, lb, ...
during
    la <- <expression (maybe using a and b)>
    lb <- <expression (maybe using a and b)>
    <produce other outputs using a and b>
end

input  la
output a
during
    a <- la
end

input  lb
output b
during
    b <- lb
end

init
    a <- <initial_val>
    b <- <initial_val>
end
```

A simple example of the use of this structure might be a for loop with a counter that determines the number of iterations of the loop. In this example, the ‘body’ of the loop will simply output the counter which decrements from the value supplied on the input (newcount) down to zero.

```
variable count, lcount

input  count
output lcount, o
    case count of
      0 then
        input newcount
        during
            lcount <- newcount
        end
    else
        lcount <- count - 1
    end
    o <- count
end

input  lcount
output count
    count <- lcount
end

init
    count <- 0
end
```

See the nanoSpa decode unit (section 5.4.2 and code in appendix C) for a more complex example of this sort of structure in practice.

Chapter 4

Using the data-driven style

This chapter contains a number of largely unrelated sections with further information about the implementation of the data-driven style and ideas about its uses.

Section 4.1 discusses the modifications made to the existing Balsa design flow to incorporate the data-driven style.

Section 4.2 discusses the back-end implementation of the new components.

Section 4.3 gives an interesting example of source-level optimisation in the data-driven style.

The possibility of automated optimisation of conventional Balsa into the data-driven style is briefly considered in section 4.4.

4.1 Integration into Balsa design flow

Figure 4.1 highlights the additions made to the Balsa design flow to support the data-driven style. (The original flow was shown in figure 2.8 on page 39.) The principal addition is the new data-driven compiler which compiles data-driven code into handshake circuits represented in the breeze format. The compilers may import breeze produced by the new compiler or the conventional compiler allowing integration of the two styles at the procedure (or part)

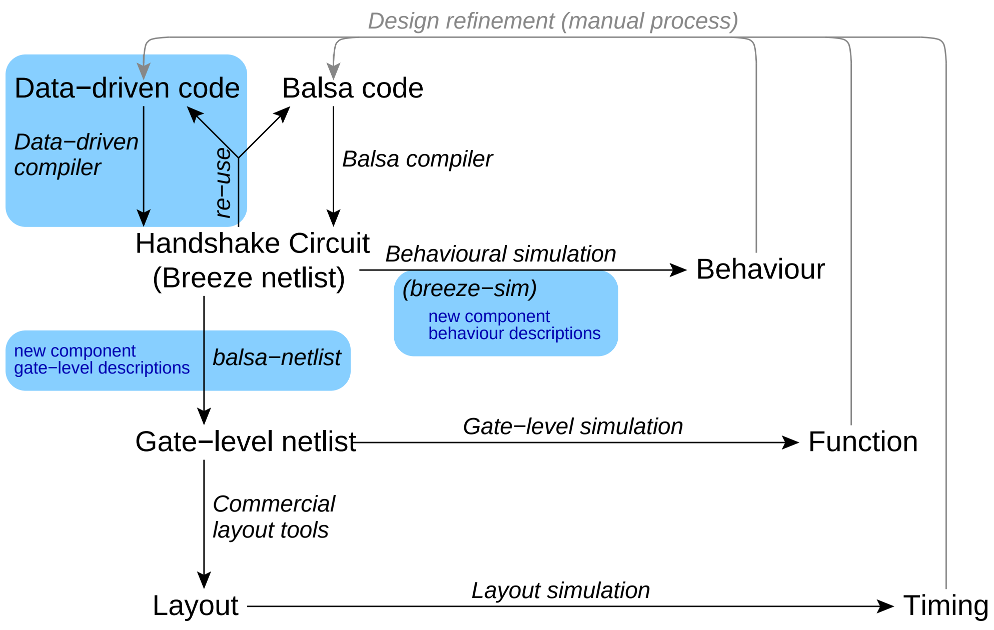


Figure 4.1: Additions to Balsa design flow

level. This makes it very easy to create designs that mix data-driven and conventional styles.

The preceding chapter introduced a number of new handshake components to support the data-driven style. The other additions to the design flow involve adding these components to the back-end and simulation environment.

The present compiler implementation allows mixing of conventional Balsa and data-driven code at the procedural level. A possible area of future work is to add support for even tighter integration of the two styles. This might present opportunities to exploit the benefits of both styles at a more fine-grained level. Data-driven and conventional Balsa code could be connected by local channels within a procedure. There is also nothing to prevent push style variables being written by conventional Balsa code. Data-driven code could then be inserted in-line with conventional Balsa to handle the 'reads' from the variable. It is not immediately clear whether this tighter integration will add value for the designer but it is an interesting area for future exploration.

4.2 Back-end implementations

In order to evaluate the data-driven style, two Balsa back-end implementations have been extended with the new components. These are the broad bundled-data and dual-rail implementations. Many existing components are re-used without modification. Pull datapath components have been re-used to create push versions where the only modification is to reverse the direction of the protocol. This means that in this implementation, the datapath logic of the two styles is the same. The new component implementations are given in appendix B.

The new components that have been added provide correct operation but may not be optimal. For example, the data-driven style uses the conventional Balsa Concur component in order to generate parallel activations. This component is implemented using T-elements to allow the output activation handshakes to complete independently. The new Dup component is implemented rather like a Concur except with data channels instead of sync channels. It is possible that these implementations may be too heavyweight in certain circumstances. If the logic of the output activations or data channels is fairly balanced, there may be little advantage in using Concur-style components to avoid synchronising between processing and RTZ phases. It is not known whether it is always safe to remove the T-elements from the Concur and Dup components. In Balsa, it is possible that one output from a Concur must complete before another can complete. Therefore synchronising between the processing and RTZ phases is not possible. In the data-driven style, it is possible that a number of concurrent activities are used to produce only a single output and the other paths are speculative and will be rejected. In this case, it would not affect correctness if the handshakes were synchronised between phases in the manner of a Fork. There is an opportunity for future work around questions of component implementation such as this.

Conventional Balsa handshake circuits did not require a reset signal to initialise components. This is sometimes considered an advantage of the approach, but is really of minimal importance. The Balsa back-end already had the ability to use a reset signal and distribute it throughout a design and the data-driven components have been designed using this feature. It would not be straightforward to implement some components such as an initialised VariablePush without a reset signal as setting the storage in the component to a particular value without a reset is problematic. In Balsa, the handshake circuit itself is used to initialise variables by sequencing variable writes to occur first, and then activating the main operation of the circuit. This solution is clearly not possible in the data-driven style and is somewhat inefficient anyway as it may lead to the introduction of a CallMux component on the Variable input which adds latency during the main operation simply to support initialisation. It is also worth pointing out that all synchronous circuits use a reset signal; it is hardly a new issue that asynchronous design has introduced.

4.3 Source-level optimisation

An often quoted advantage of Balsa is that, due to the direct compilation, the designer is able to optimise for performance, area, or power consumption at the source code level. The direct compilation of the data-driven style allows for very much the same thing.

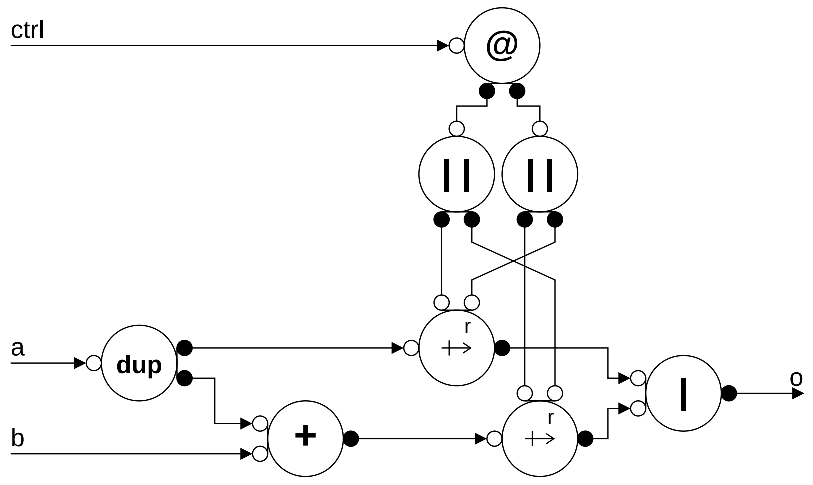


Figure 4.2: Unbalanced speculative operation (A)

4.3.1 The example

The following data-driven description (design **A**) will be used to demonstrate this by example.

```
-- design A
input  ctrl, a, b
output o
during
    case ctrl of
      1 then
        o <- a       -- pass
    else
        o <- a + b   -- add
    end
end
```

This code writes either a or the sum of a and b to the output o depending on the value of ctrl (which will be assumed to be 1 bit). These two alternative operations will be called *pass* and *add*. The add operation is a convenient choice for the slower operation as the delay can be easily modified by adjusting the inputs to alter the length of the carry chain (the default adder used is a basic ripple-carry adder). Figure 4.2 shows the resulting handshake circuit.

In section 3.5.4 it was noted that a description such as this, where one of the conditional operations is potentially much slower than the other, could lead to reduced throughput, particularly if the pass operation is frequently selected. Additionally, if energy consumption is a consideration then speculating on the add operation when it is not required is likely to increase the energy requirement of the circuit. This is especially true in delay-insensitive implementations where, as the request is encoded with the data, if no add takes place then no transitions will occur in the adder.

The graph in figure 4.4 shows the results of simulating this circuit using the dual-rail back-end and unit gate delays. The test-bench used for the simulation has zero delay so the inputs are supplied as quickly as they can be accepted and the output is consumed as quickly as it is produced. The width of the adder is 16 bits. It can be seen in the graph how the cycle time for pass operation increases as the length of the carry chain increases due to the speculative add operation being performed, even though it is not required.

Note that as the environment has zero delay, the results here represent the worst possible case in terms of the impact of the speculative operation. If the data on channels a and b were to arrive earlier than that of ctrl then the impact of the speculation on the throughput might be reduced.

4.3.2 Avoiding speculative operation

Either from a desire for throughput, or reduction in energy use, it may therefore be desirable to avoid performing the slow operation (add) unless it is actually required. The description given on the next page (design **B**) performs the same operation but does not speculate on the add operation. The addition is now contained within its own block. The control is used to supply the inputs to the add operation only when the result is actually required. The result is then only produced when required so it becomes a conditional input to the block that generates o.

```
-- design B

channel addA, addB, pass
channel addR
channel ctrl0
channel ctrl1

input  addA, addB
output addR
during
    addR <- addA + addB
end

input  ctrl
output ctrl0, ctrl1
during
    ctrl0 <- ctrl
    ctrl1 <- ctrl
end

input  ctrl0, a, b
output addA, addB, pass
during
    pass <- a
    case ctrl0 of
      0 then         -- supply the operands for the
        addA <- a    -- addition only when they are
        addB <- b    -- required
    end
end

input  ctrl1, pass
output o
during
    case ctrl1 of
      1 then
        o <- pass
    else                     -- the add result is
        input addR during    -- only produced
            o <- addR        -- when required
        end
    end
end
```

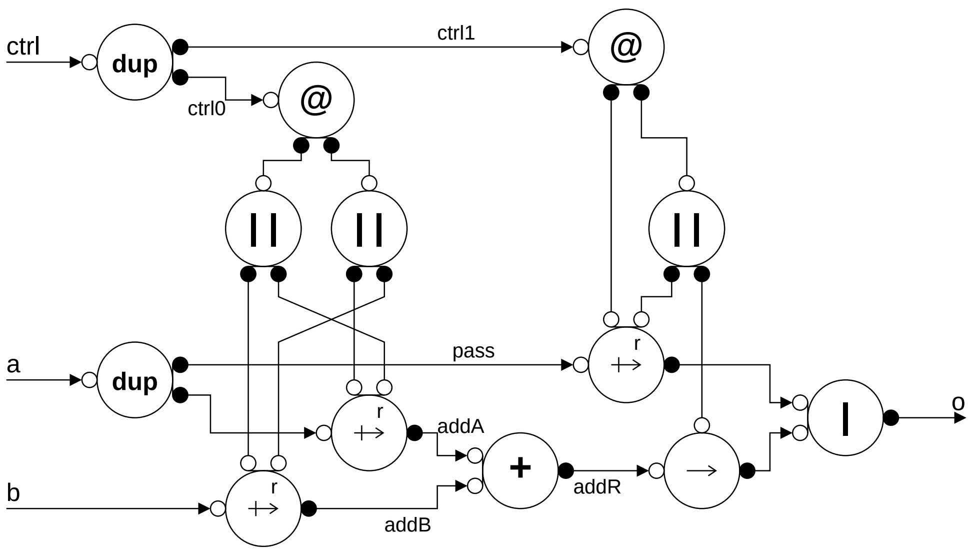


Figure 4.3: Speculative operation avoided (B)

See figure 4.3 for the circuit produced for design B. As well as increasing the latency of the add operation, the additional drawback of this design is that the area is a little larger than the original. See table 4.1 and figure 4.5 for a comparison of the area. The clear advantage of design B, as shown in figure 4.4, is that when doing the pass operation the throughput is significantly improved and constant irrespective of the latency of the slower operation.

Which design is then to be preferred, A or B? Clearly this depends on the difference in the delays of the two operations, the specific requirements of the designer and the anticipated usage pattern of the circuit (i.e. how many add operations are performed compared with pass operations). Assuming the average delay of the slow operation is the add with a carry chain length of 4 then the graph in figure 4.6 shows that if in approximately 75% or more cycles the add operation is selected, then design A provides better throughput overall. However, if the add operation is selected less than 75% of the time then design B will have better overall throughput.

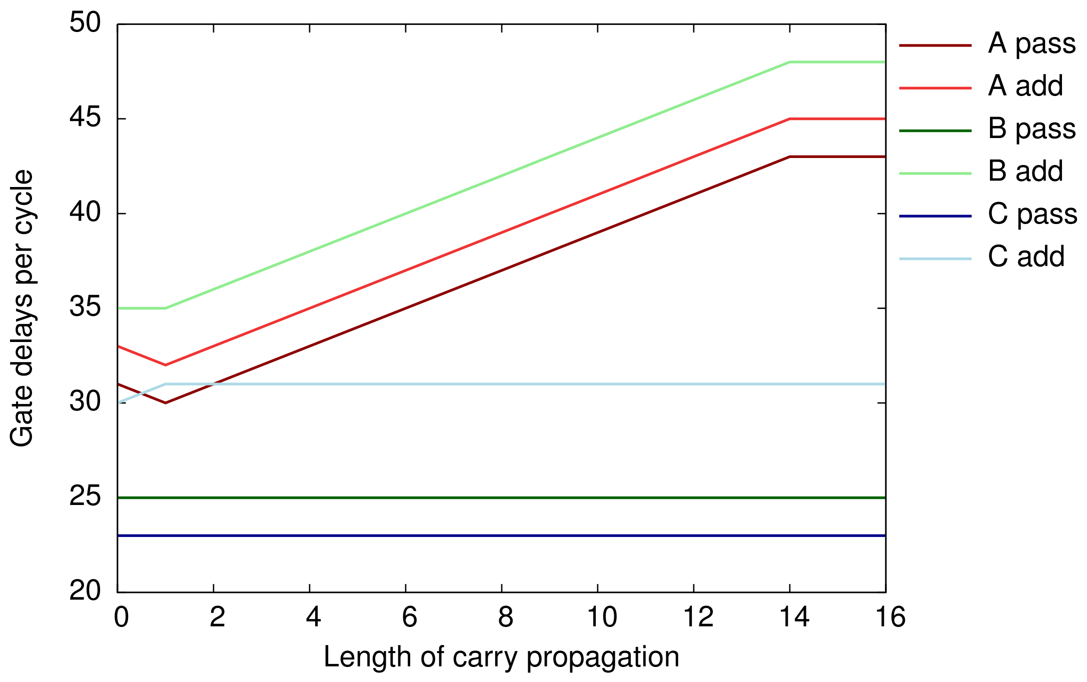


Figure 4.4: Source-level optimisation example results

4.3.3 Adding pipelining

What if the best of both worlds is required? Can anything be done to further improve the throughput? As design B contains two stages of operation, there is an obvious opportunity to increase the throughput by pipelining these two stages. Unlike the control-driven style of conventional Balsa, in the data-driven style it is straightforward to add pipelining. By using design B but changing the channels ctrl1, pass, addA, and addB to variables the circuit in figure 4.7 (design **C**) is produced. The only changes to the code for design B are in the first five lines where some channel declarations are changed to variables:

```
-- design C

variable addA, addB, pass
channel  addR
channel  ctrl0
variable ctrl1
```

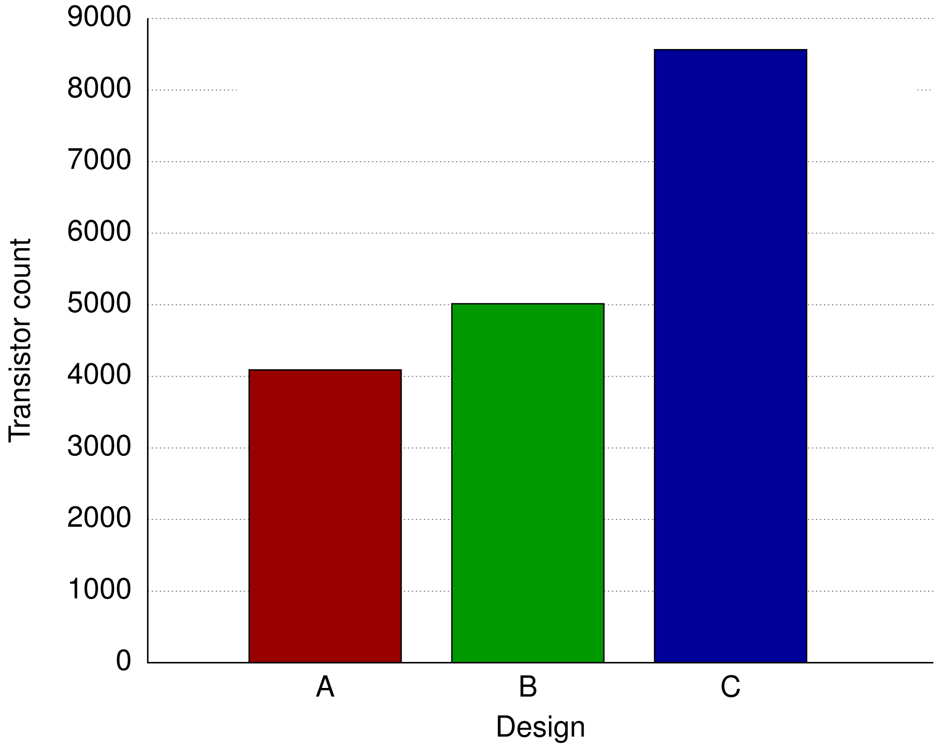


Figure 4.5: Source-level optimisation example area

Figure 4.4 shows how the pipelining improves the throughput of both operations in comparison to A and B. Figure 4.6 confirms that design C provides better throughput regardless of the pattern of usage of the two operations. The cost of the new design is the extra VariablePush components which significantly increase the area over that of designs A and B (see figure 4.5).

Design	Pass gates/cycle	Add gates/cycle	Transistor count
A	33	35	4090
B	25	38	5015
C	23	31	8563

(carry chain length: 4)

Table 4.1: Source-level optimisation example results

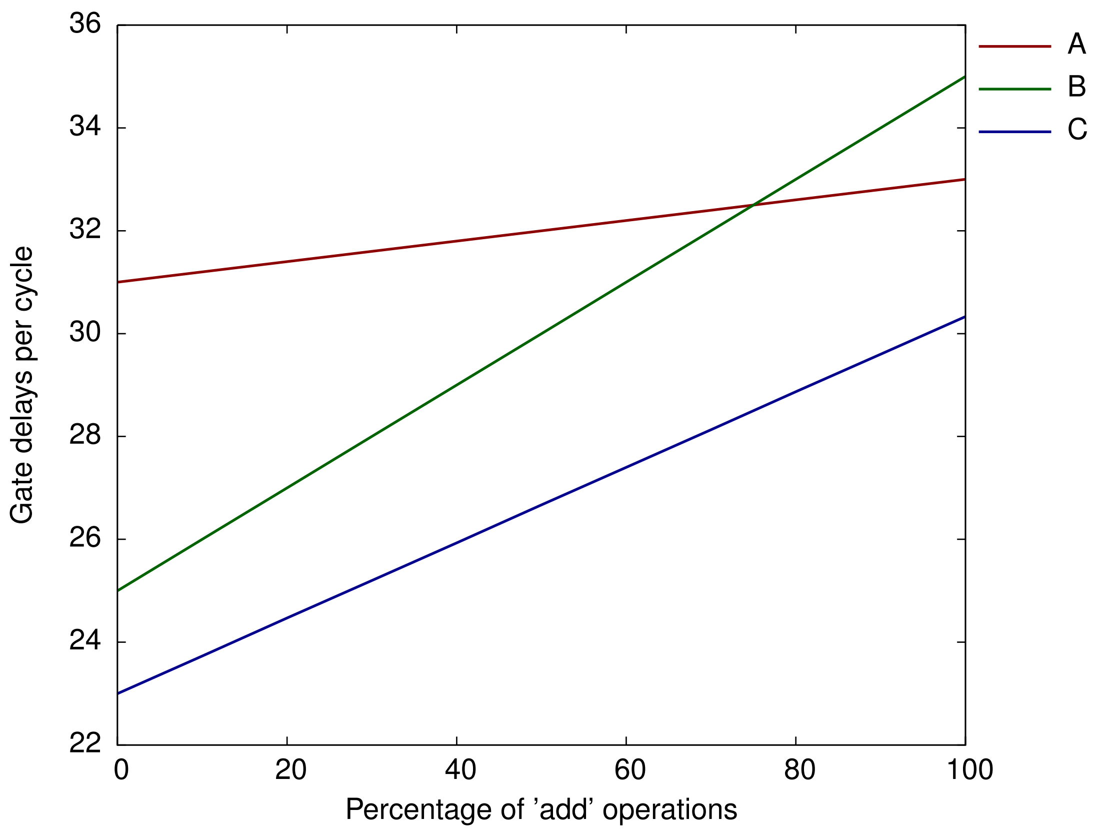


Figure 4.6: Comparing the designs

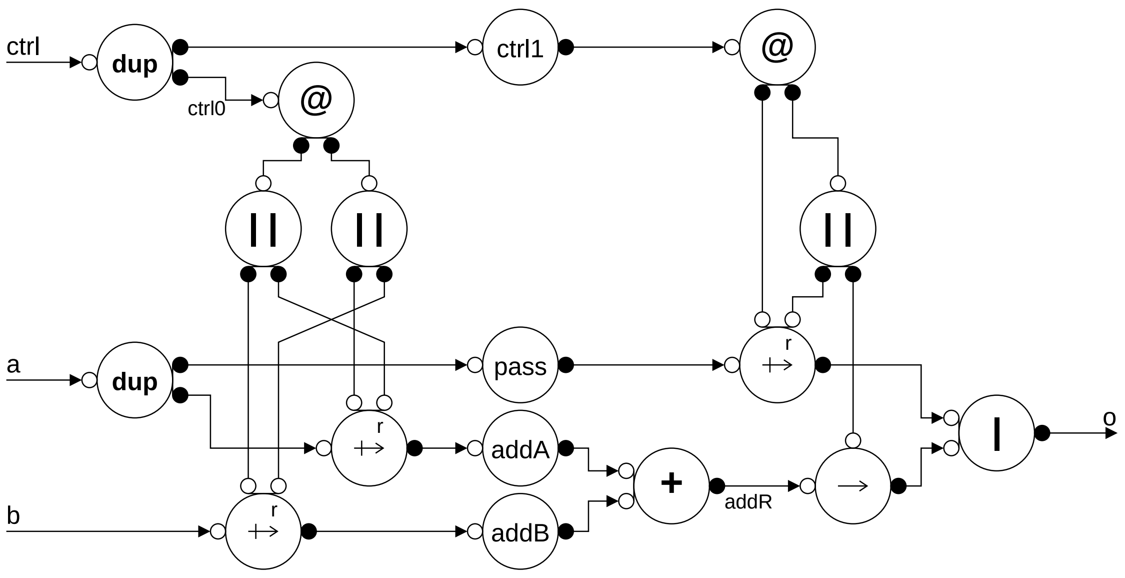


Figure 4.7: Extra pipelining added (C)

4.3.4 Discussion

This example is not intended to suggest that a designer would, or indeed should, use an analysis of this depth for every module in their design. In many cases it may be that knowledge of the anticipated operation is sufficient to intuitively make design decisions. For instance, in this example, if it is known that the add operation will be uncommon then designers with some experience should realise that they should avoid supplying operands to the operation except when it is definitely required. In general, it is difficult to draw conclusions on what may or may not be intuitively obvious to any given designer. Exactly the same statement may of course be made with respect to conventional Balsa. The claim being made here (if any) is that the data-driven style benefits from the same advantage that is claimed for conventional Balsa. That is, that the direct compilation allows a designer to make source-level optimisations that will have a predictable effect on the resulting circuit. At the very least, the benefit of a high-level synthesis approach is that it allows for rapid prototyping and testing of alternative designs where it is not immediately clear which is preferable.

4.4 Data-driven style as a target for optimising Balsa

The work in this thesis resulted from looking at conventional control-driven Balsa handshake circuits and attempting to overcome the overheads inherent in the control-driven structure. This result is a new style of handshake circuit and a new description language in which to describe these structures. Another possible approach that was examined was to automate transformations to existing handshake circuits to produce more efficient structures, similar to those of the data-driven style. This work was abandoned, in large part, due to uncertainty as to what the resulting structure of the transformations should be. The data-driven style was subsequently developed, in part, to explore what the possibilities and limitations of such a style might be.

The possibility of using the data-driven style as a target for optimisations to Balsa circuits now presents itself as an area for future work. A requirement of any optimisation strategy is that while the internal behaviour of a circuit may be modified, the external behaviour should remain the same. Any environment in which the original circuit was deployed should be able to take advantage of the optimised version without requiring any modification.

This requirement is not easy to meet as understanding the behaviour of a large design made up of many communicating processes is not trivial. Existing approaches do exist for generating data-driven style circuits from CSP-like descriptions. For example, the Caltech synthesis tools which use data-driven process decomposition [WM01] to decompose a sequential program written in CHP into a number of smaller processes. However, this decomposition is only correct where the design conforms to particular requirements. A similar approach using Haste as the input language is being developed as part of the CLASS project [CLA]. Both of these techniques involve pipelining sequential operations which will result in different external behaviour.

For example, consider an environment that has two ports, one for control and one for data. The environment requires that control is communicated followed by data and the data communication must be completed before another control communication occurs. A Balsa description structured similarly to the following might be used to implement this interface:

```
loop
    <generate C and V>
    ;
    <generate D using V>
end
```

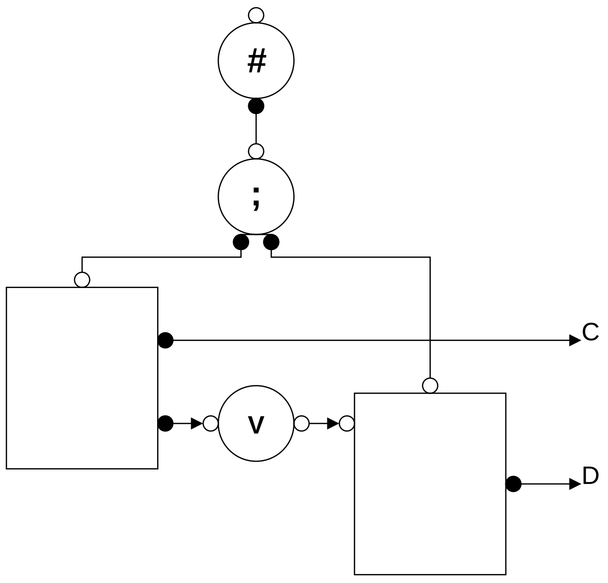


Figure 4.8: Balsa process example

The handshake circuit structure is shown in figure 4.8. It is clear from this structure that a communication will take place on C, followed by any communications on D before returning to C. This process might be 'decomposed' along the lines of the methods mentioned above into the following two data-driven processes (see figure 4.9):

```
output C, V
during
    <generate C and V>
end

input  V
output D
during
    <generate D using V>
end
```

These two processes do not necessarily behave in the same manner as the original. The order of communications may now feature a second communication on C, during the communication on D. This is because the explicit sequencing in the control-driven process is not replicated in the decomposed

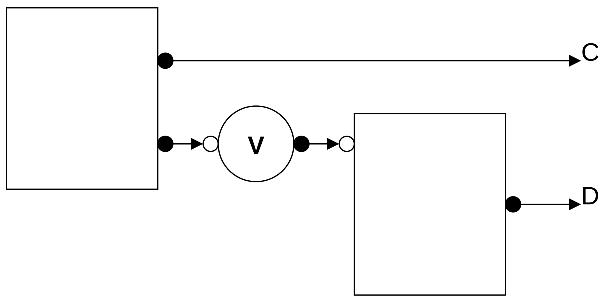


Figure 4.9: Data-driven process transformation

processes. The first process could produce a second communication on C even though it is blocked writing to V while the second process uses it, whereas the Sequencer prevents this in the Balsa implementation.

It is possible to give the programmer responsibility for ensuring that the behavioural changes made by any optimisation scheme do not create errant operation. This is the approach adopted in data-driven process decomposition of CHP, where the programmer must ensure the program is 'slack elastic' [MM98]. It is unclear how this complex requirement might be communicated to a user who is not already familiar with the techniques being employed in transforming the program. As automated process decomposition does not produce results that are nearly as effective as can be achieved manually [WM03], an expert user may prefer to design in a data-driven style anyway. In this case, the data-driven language introduced in this thesis should prove more applicable than conventional Balsa.

Despite these reservations, some method along the lines of process decomposition could prove very effective in optimising Balsa descriptions. As an automated optimisation approach it is important that the resulting circuit must continue to do what the designer originally wrote with respect to the environment.

A possible area for future work is applying some analysis to determine the external behaviour of a given circuit. This can then be used to ensure that optimisations preserve the correct behaviour; or to generate a wrapper that ensures the correct external behaviour is preserved, while allowing optimisation of the internal workings of the circuit. Determining the behaviour could prove a challenging problem. It is relatively straightforward to see what the possible behaviour of a single process in isolation is. It is not so straightforward to see what the external behaviour of a number of communicating processes might be.

Adding assertions to the design that would allow optimisations to take advantage of knowledge of the environment in determining permissible modifications to behaviour is also an interesting possibility for future work.

Chapter 5

Design Example

The last two chapters introduced the data-driven handshake circuit style and description language. This chapter will attempt to demonstrate the benefits and drawbacks of the style by means of a large design example. The example design, known as nanoSpa, is a 32-bit microprocessor which implements what is essentially a slightly cut-down version of the ARM instruction set[SJ00].

5.1 Manchester Asynchronous ARMs

The nanoSpa is part of a series of asynchronous ARM implementations developed by the AMULET (latterly, APT) Group at the University of Manchester. The first three processors were named AMULET and developed primarily using hand designed logic [FDG+93, FGT+97, FGG98].

The final processor was named SPA[PRB+03] and was fully synthesised using Balsa. The objective of the SPA project was to investigate asynchronous logic as a means of increasing resistance to differential power analysis (DPA) in a smart card application. Performance was therefore not a major requirement for the application or the objectives of the project which turned out to be just as well.

The handshake components implementations, and architecture of the SPA were designed to operate in such a way that a balanced power usage would be seen regardless of the data being operated on. For this reason, the performance was deliberately impaired as all logic was designed to take worst-case time rather than average.

However, even considering the security features, the performance of the SPA was somewhat disappointing. This can mainly be attributed to the inexperience of the designers in using Balsa and the limited time available due to the project imposed deadline for production of the chip. It should be pointed out that the SPA was 100% ARM compatible and operated almost flawlessly first time in silicon in single and dual-rail implementations, both entirely synthesised from the same Balsa source description.

Since the SPA, and using the experienced gained therein, the nanoSpa has been gradually developed with the sole objective of making a Balsa synthesised asynchronous ARM of the maximum possible performance. Development has not reached the stage where the processor implements the entire instruction set but most of the main features are present and benchmark programs can be run in simulation to produce a good idea of the performance (which is almost ten times that of the original SPA). This makes it an excellent example in demonstrating whether a data-driven circuit can offer performance improvements over the best available conventional Balsa circuit.

5.2 Objectives of this example

1. To demonstrate that the data-driven synthesis flow can be used to construct a significant design.
2. To compare the performance of a high performance Balsa design with the closest possible equivalent in the data-driven style.
3. To demonstrate the integration into the existing Balsa design-flow and the use of mixed Balsa and data-driven designs.

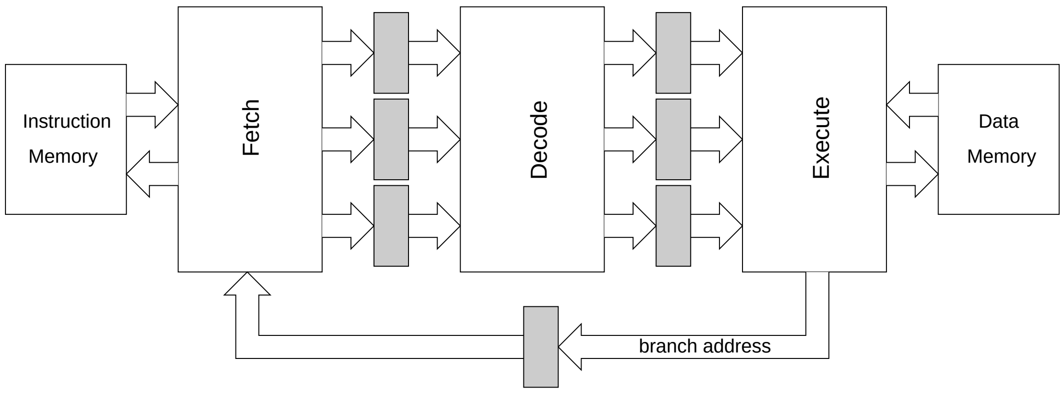


Figure 5.1: nanoSpa Pipeline

4. To attempt some level of qualitative comparison between the features and flexibility offered to the designer in both description styles. In particular, it is believed, that this example demonstrates that the data-driven description differs very little from the style of Balsa code that an experienced Balsa developer would write. Indeed, converting the Balsa nanoSpa into a data-driven description provided very few challenges.

5.3 The nanoSpa

nanoSpa utilises a Harvard architecture and a classical three-stage ARM pipeline; the stages being fetch, decode and execute. Figure 5.1 shows this basic structure.

Unfortunately this does not lead to a balanced pipeline and the complexity of the execute stage causes it to dominate the performance of the overall processor. However, extending the length of the pipeline has proved difficult to achieve in Balsa due to the increased requirements on the design of the register bank. This will be further elaborated in section 5.4.3.

The description consists of approximately 3000 lines of Balsa. The following ARM features have not been implemented at the time of writing. This is due to time constraints rather than any anticipated difficulty in implementing these features.

- Multiply instructions
- RRX shift operation
- Half word/signed byte load/store
- Only two operating modes are implemented: system (privileged mode) and user.
- Memory protection
- Interrupts.
- Memory aborts.
- Thumb compressed instruction set.
- Co-processor interface

5.4 Data-driven nanoSpa

The data-driven nanoSpa has been written by the author in the data-driven input language. The description is roughly the same length as the Balsa original. As far as possible, the micro-architecture of the processor has been precisely copied from the Balsa description. As a consequence, most of the synthesised datapath logic is the same as the Balsa nanoSpa, and the control contains most of the significant differences. The intention is to attempt to explore the advantage gained by using the data-driven style in describing a design that is as close as possible to a Balsa description, rather than by tailoring the design specifically to suit the data-driven style.

The two major exceptions where it was necessary to make significant changes to the architecture are in the decode unit, due to its use of (temporal) iteration, and the register bank, due to its reliance on Balsa-style variables. These issues will be discussed in more detail in the appropriate sections below.

Code for selected modules of the data-driven nanoSpa is presented in appendix C.

5.4.1 Fetch

The fetch unit is relatively small compared to the other two pipeline stages and it is therefore just about possible to view the entire handshake circuit in one figure. Figure 5.2 shows the original Balsa generated handshake circuit and figure 5.3 shows the circuit generated from the data-driven description.

The most problematic aspect of the fetch unit is interrupting the default sequential fetching of instructions with a new program counter value from a 'branch'[1] instruction. As in synchronous implementations, pipelining means that additional instructions after the branch will be fetched before the branch is actually executed. This branch 'shadow' is deterministic in a synchronous design but in an asynchronous implementation the number of additional instructions fetched is often non-deterministic.

In nanoSpa this non-determinism is handled using the so-called *colour* mechanism, originally used in the AMULET processors. Each instruction fetched is associated with a one bit colour. The execute unit maintains the current operating colour of the processor. When a branch instruction is executed the colour is inverted and the fetch unit is informed of the change of colour. The colour of subsequent instructions fetched in the shadow of the branch will not match the colour in the execute unit and are therefore discarded. When the fetch unit updates the program counter following the branch, the subsequent (correct) instructions are associated with the new colour.

Unfortunately, if the number of instructions fetched in the branch shadow

[1]note that in ARM many instructions in addition to the actual branch instruction may be used to update the program counter and are here considered as branch instructions.

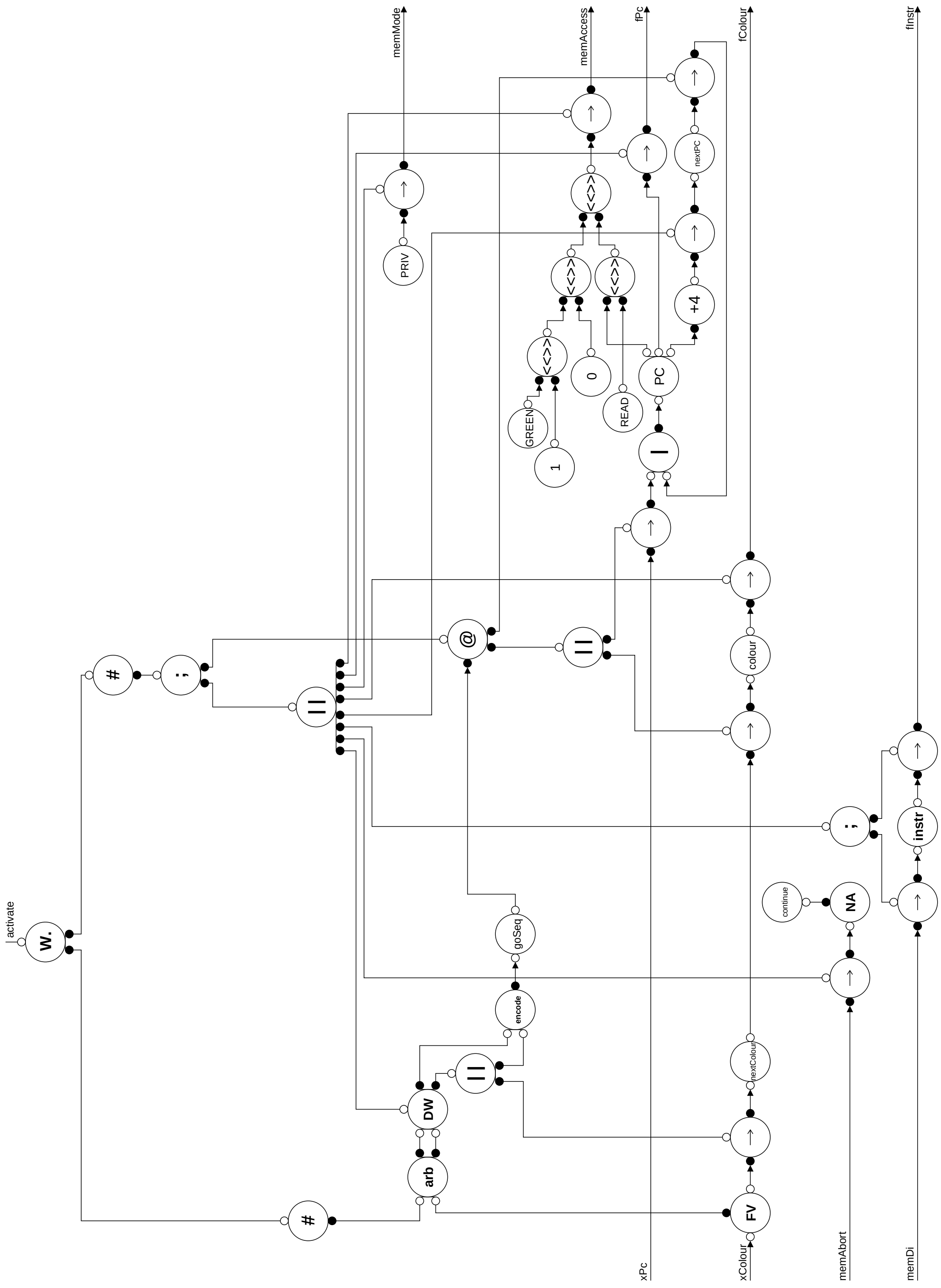


Figure 5.2: nanoSpa fetch Balsa handshake circuit

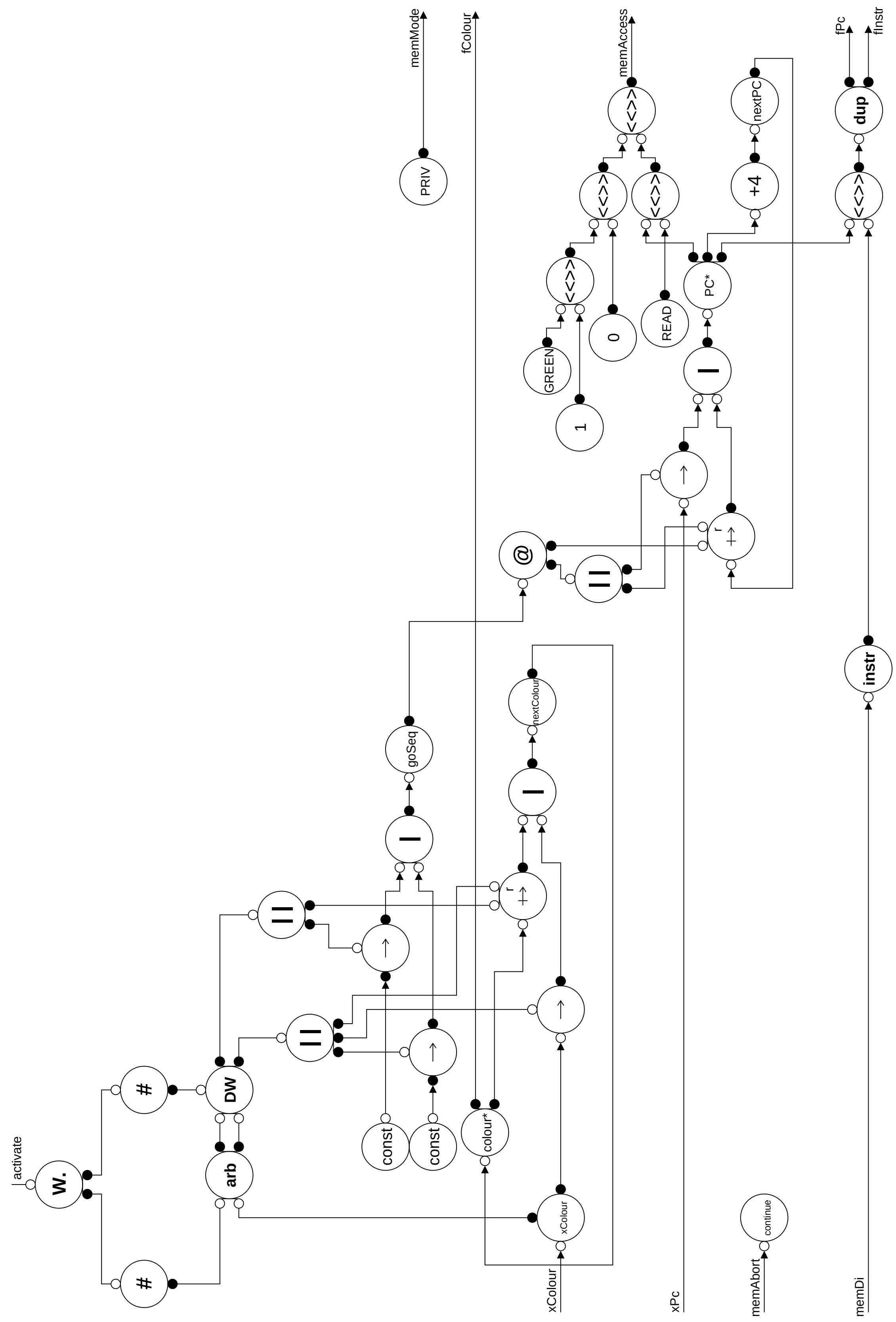


Figure 5.3: nanoSpa fetch data-driven handshake circuit

is too great then the overhead of discarding them can have a significant effect on the performance of the processor. This problem is manifest in both implementations of nanoSpa but to a larger degree in the data-driven version. As the cycle time of sequential fetches is improved in the data-driven version, the number of instructions in the branch shadow is frequently also increased. As these instructions require extra time to be discarded, the overall performance of the processor is actually reduced by improving the fetch unit!

There are two obvious methods of reducing the impact of this problem.

1. Decrease the overhead for discarding instructions. At the time of writing, several architectural features are currently being examined to reduce the overhead of discarding instructions but these were not developed sufficiently for inclusion in the thesis. In any case, modifications to the architecture are outside of the scope of this thesis.

2. Somewhat ironically, the second solution is to deliberately make the fetch unit slower in order to obtain a more balanced pipeline and reduce the average length of branch shadows. Essentially this means making a trade-off between between the throughput of the sequential operation and the latency of interrupting this operation with a branch. The optimal solution in this trade-off is to some extent dependent on the program which is running. In an extreme case, a program containing no branches would suffer no impediment from using the fastest possible sequential throughput. On the other hand, a program consisting entirely of branches would be most efficient if the fetch unit was so slow as to prevent there being any branch shadow at all.

In order to provide as fair a comparison as possible between the two versions of nanoSpa the following approach has been adopted:

Where the fetch unit is being considered in isolation, the original full speed fetch unit will be used. Where the fetch unit is being used as part of the processor, delays in the sequential fetch operation will be introduced to both designs.

These delays will be manually tuned so that the version in question delivers the maximum performance when running the Dhrystone benchmark program.

The need for this manual intervention is a product of architectural deficiencies, not the synthesis method. The data-driven style will, by improving the performance of the fetch unit, decrease the overall performance by lengthening the branch shadow. The most equitable comparison that can be readily achieved for the overall processor is to tune each implementation so it achieves its best possible performance for the same program. In any case, it is probably more instructive to compare the performance of individual units in the design thereby reducing the impact of architectural issues, but some indication of the overall performance is also desirable.

5.4.2 Decode

Unusually for a RISC-style processor, the ARM instruction set contains support for multi-cycle load and store instructions. These load and store multiple (ldm/stm) instructions allow any given subset of registers to be loaded from or stored to contiguous words in memory using a single instruction. The nanoSpa implements these instructions in the decode stage by simply generating and issuing multiple single memory transfer operations to the execute unit. The general structure used is illustrated in figure 5.4. A huge case construct is used to select either the decode for regular instructions, or the ldm/stm iterative decode. As both sides of the case are not activated concurrently, they can both produce the same outputs destined for the execute unit. The Balsa compiler generates CallMux components to merge equivalent pairs of outputs into a single channel.

Attempting to replicate this structure presents some difficulty in the data-driven style. The iterative decode for ldm/stm instructions makes use of the Balsa while loop structure to repeatedly generate memory transfer operations.

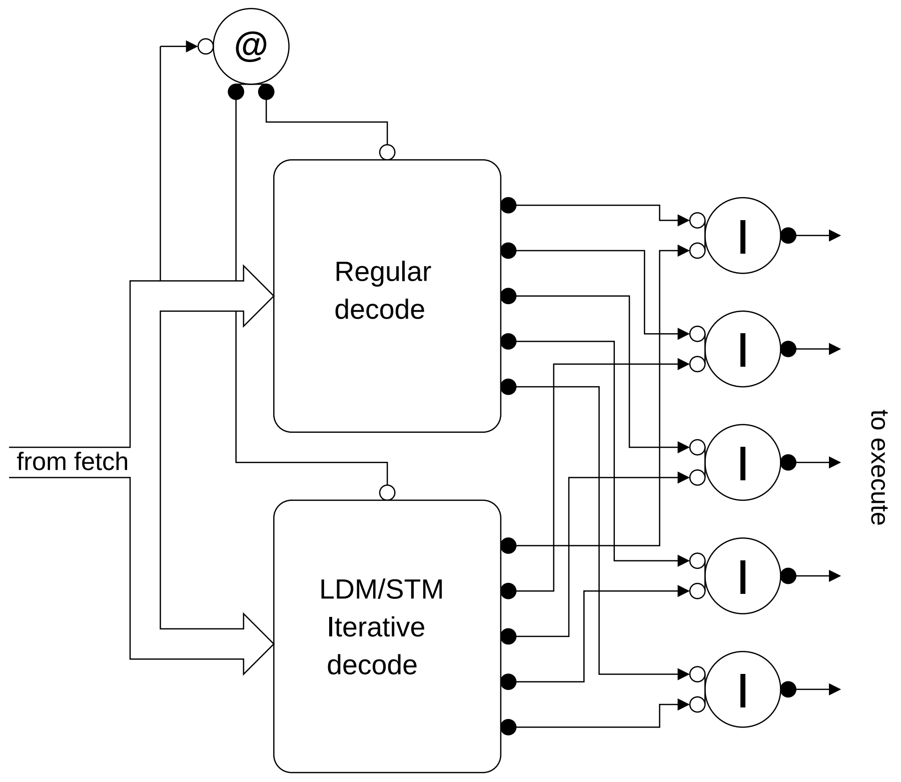


Figure 5.4: nanoSpa decode structure

In the control-driven style the handshake for the inputs to the decode can enclose all of this iterative operation allowing the inputs to be read repeatedly by each iteration.

An iterative structure of this nature is not available in the data-driven style. Providing more language support for generating this sort of operation is a subject for future work. However, it is quite straightforward to re-arrange the structure of the decode, based on the example given in section 3.7 (page 111), to implement the multi-cycle instructions as shown in figure 5.5. In this structure the ldm/stm decode is no longer itself iterative. Instead the whole decode can be viewed as iterative with regular instructions simply being a special case requiring only a single iteration. When an instruction arrives at decode it is passed through the multiplexor to the decode logic. If the instruction is an

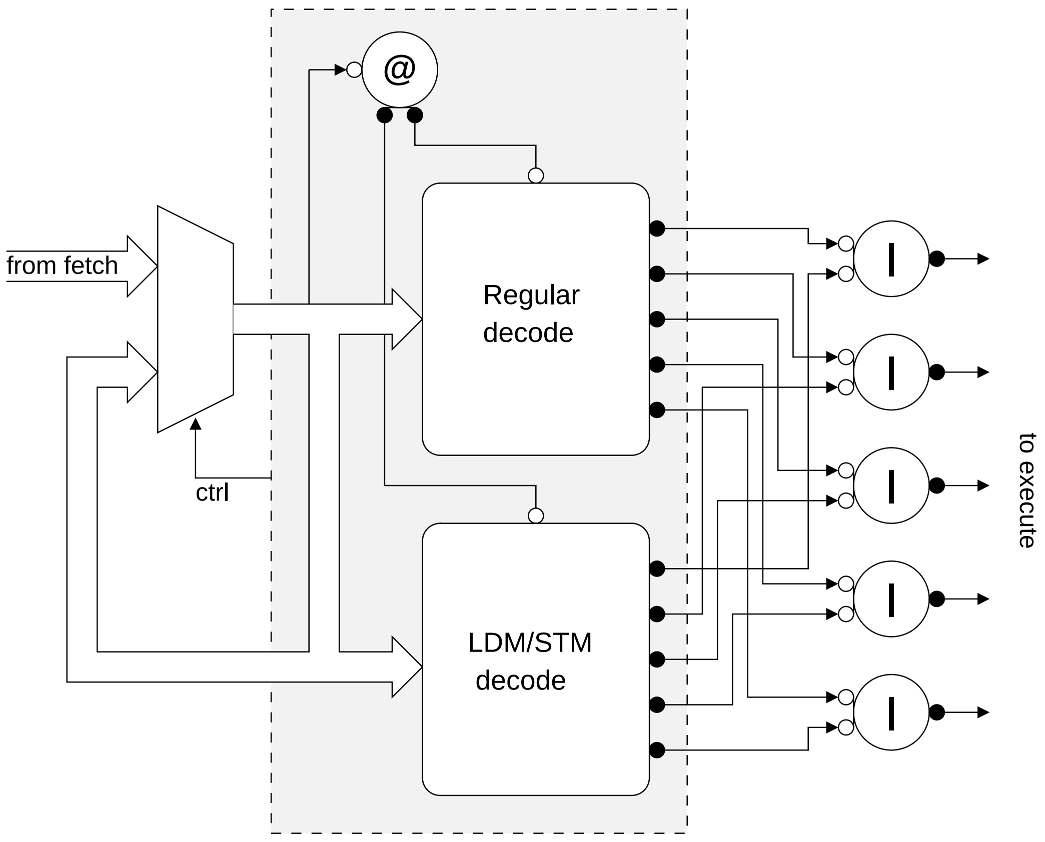


Figure 5.5: nanoSpa data-driven decode structure

ldm or stm the necessary data for the next iteration is passed back to the multiplexor and the control signal is set so as to re-inject the data as the next instruction. When the ldm/stm is finished, or after a single 'cycle' if the instruction is a regular instruction, the multiplexor is signalled to inject the next instruction being sent from fetch.

Although this may not be the most efficient implementation, it has the important advantage that the two decode blocks (for regular or ldm/stm instructions) which basically consist of a large number of case structures (one for the generation of each output) can be translated directly to the data-driven description. This means that the code which generates the grey shaded area on figure 5.5 (i.e. the bulk of the decode unit) was basically copied directly from the Balsa description without any significant modification saving a great deal

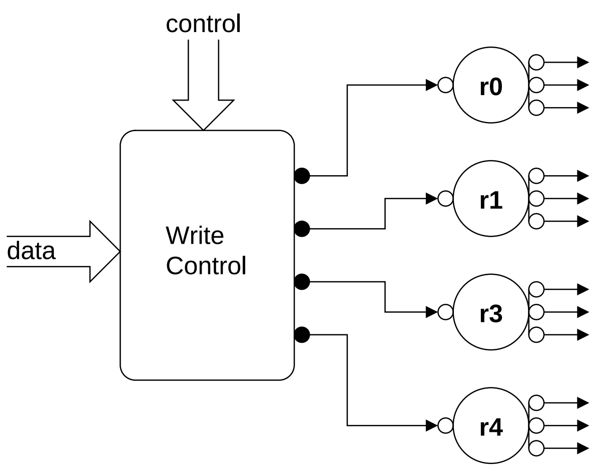


Figure 5.6: Balsa nanoSpa register write structure

of time and effort.

Unfortunately, the overhead of this decode structure means the performance improvements for regular decodes are not as large as those seen in some other modules. However, it is still sufficiently fast as to have no overall effect on the speed of the entire processor. The new structure does significantly improve performance for multi-cycle decodes.

5.4.3 Register Bank

In the three-stage ARM pipeline the execute stage reads the register bank, processes the data and then writes back to the register bank. Three read ports are required as several ARM instructions require three operands. Load instructions may write-back to two registers, one to load the register from memory and one for an addressing mode with write-back. These two writes are sequenced in nanoSpa so only a single write port is required.

As the register bank is contained in a single pipeline stage, it is possible to sequence the three reads (which occur in parallel) with the two writes. In order to lengthen the pipeline it would be necessary to split the read and write

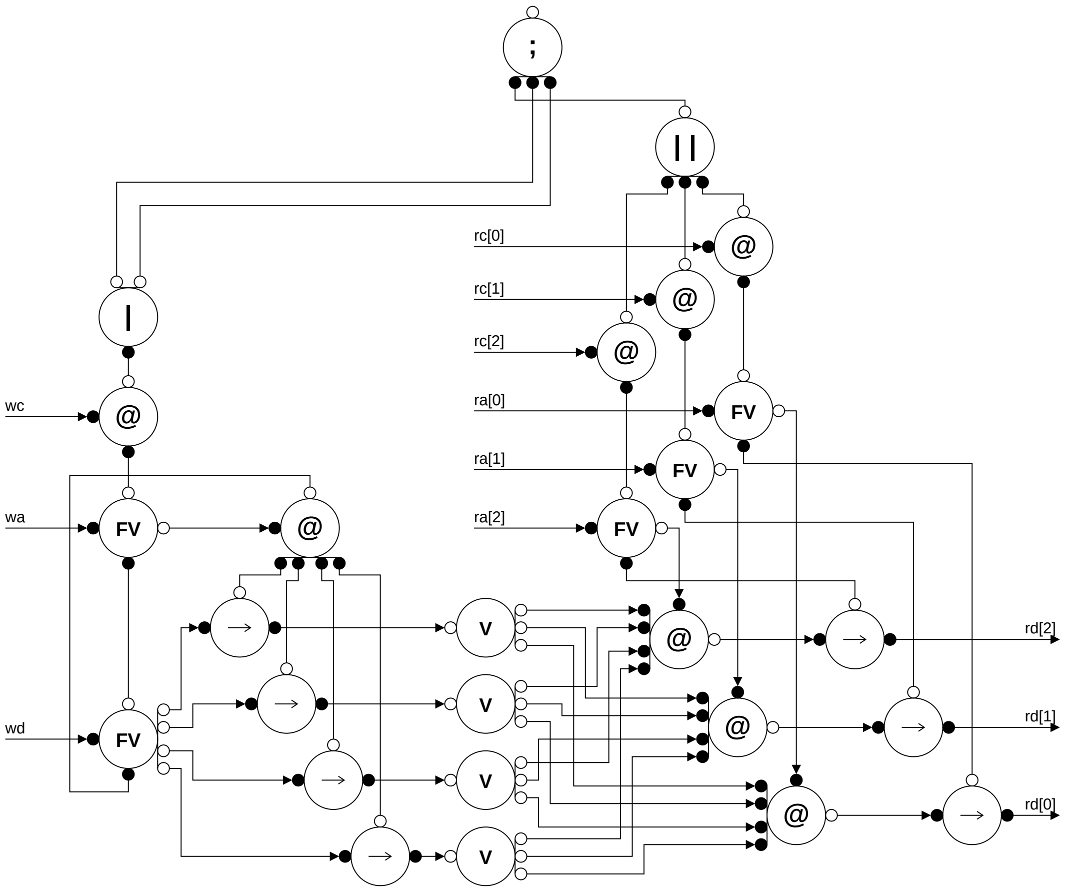


Figure 5.7: Simplified nanoSpa Balsa register bank

phases of the register bank so that they can occur in parallel. It is also necessary to employ some mechanism to ensure the correct value is read from a register on which there is a write outstanding (to avoid write-after-read hazards). In the AMULET processor series this was achieved first by register locking [PDF$^+$92] and then by a re-order buffer[GG97]. No method has yet been found to efficiently describe these mechanisms in Balsa or the data-driven style and no alternative has yet been proposed.

The Balsa nanoSpa register bank uses the general read and write structure for variable arrays discussed in section 3.5.8 (page 94). The passive ported

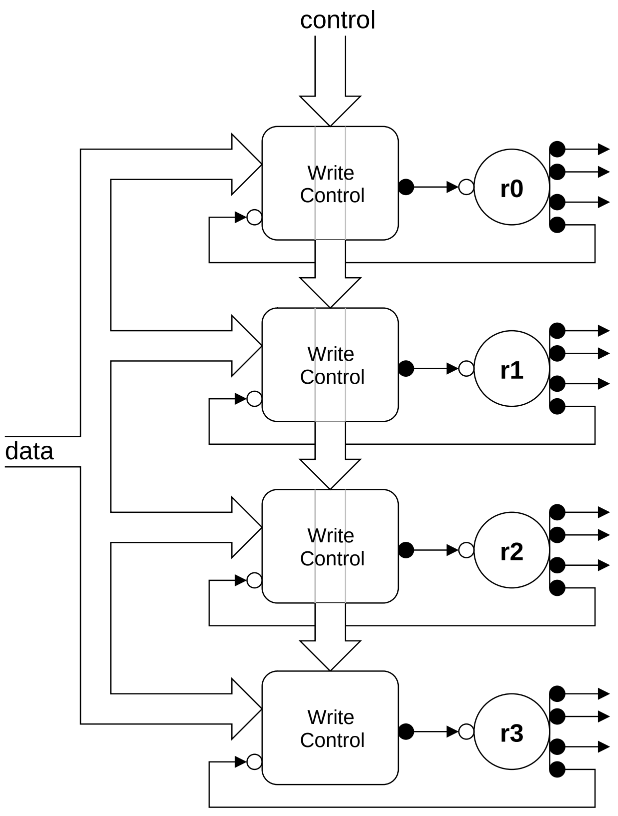


Figure 5.8: Data-driven nanoSpa register write structure

Variable component allows reads and writes to occur to variables in any arbitrary order. Figure 5.6 attempts to illustrate the structure for writing to the registers in the Balsa register bank. (Only four registers are shown.) A single write control block is used to steer the incoming write data to the appropriate register. The read control is similarly straightforward; each read port simply pulls the required data from the required register. Each 'cycle' of operation in the register bank consists of a read phase where up to three values can be read in parallel on three read ports, and a write phase where two writes can occur. See figure 5.7 for a simplified handshake circuit of the Balsa register bank.

As discussed in section 3.5.8 (page 94), when using push style variables it is not so easy to provide this general structure. In order to read from any variable, it is necessary for that variable to push its data. Therefore, in order to implement the register bank in the data-driven style it is necessary to write to

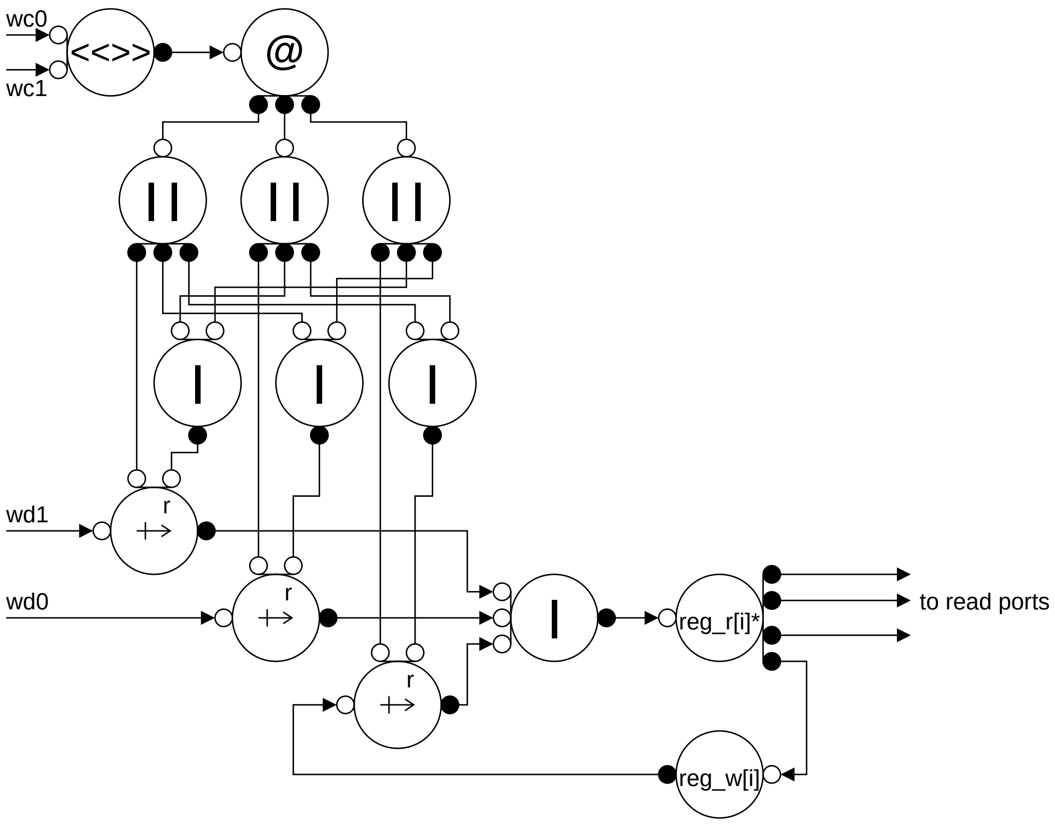


Figure 5.9: Data-driven register 'cell'

every variable (i.e. register) during every cycle. The data-driven register bank write structure is illustrated in figure 5.8. The write control and data are here duplicated to individual write control units belonging to each register. These individual units decide whether to write the data to their respective register. If they do not write the data, they recycle the existing value and write this to the register instead. The subsequent read phase may therefore pick the appropriate data from any register as all registers will push data.

The data-driven register bank structure results in an individual 'cell' for each register that controls the writes to that particular register (see figure 5.9 for the handshake circuit). A 'read unit' is generated for each read port (see figure 5.10). This structure results in improved performance but also significantly increases the area over the Balsa counterpart. It will also significantly

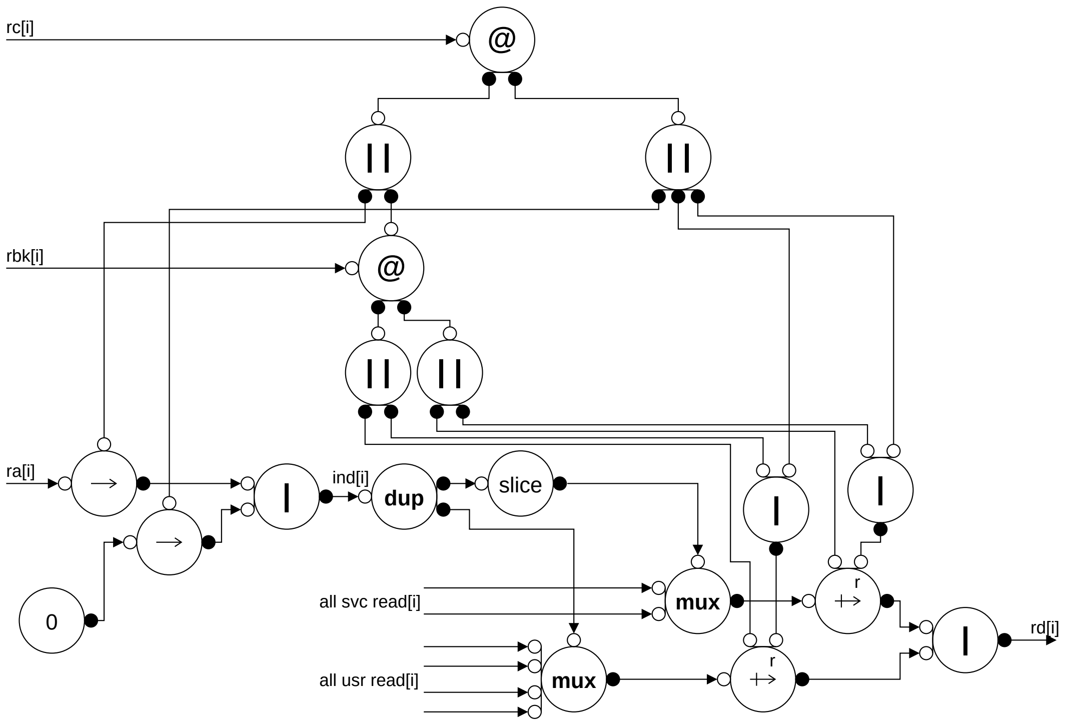


Figure 5.10: Data-driven register read 'unit'

increase the energy consumption as every register is written on every cycle.

5.4.4 Execute

The execute unit is complex and a description of even moderate detail is beyond the scope of this thesis. The unit is made up of a number of small modules that operate independently. Some modules process data while others 'steer' and merge the data to direct it around the modules as required to execute the decoded instruction. Control inputs to these modules come either from the decode unit, or from the ExecuteControl unit. The function of a few modules is briefly summarised below. These modules include the main functional modules and examples of a steering and merging module.

ALU

The ALU is fairly self explanatory. Depending on a control input, it performs addition, subtraction, a logical operation, or a move. It also calculates new values for the processor status flags based on the result of the operation (Carry, Overflow, Zero and Negative).

ExecuteControl

In the ARM instruction set every instruction is conditionally executed. The ExecuteControl module is responsible for determining whether each instruction will be executed or not. This decision is based on the condition code of the instruction, the status flags of the processor and on the colour of the instruction and the current operating colour. (See section 5.4.1 on the fetch unit for an explanation of the colour mechanism.)

ExecuteControl produces a number of outputs that are used to determine the operation of other units in execute. Precisely which outputs are produced and their values is dependent on the instruction type and whether it is to be executed or not.

mux3

This simple module is used to merge inputs to the ALU which may come from either the register bank, an immediate value or from a feed back loop from the previous result of the ALU (to support load and store multiple instructions). The module has a control input that determines which of its three inputs is to be forwarded to the output.

steerDi

This module is used to direct data being read from memory to the correct destination. If the destination of a load is the program counter then the data must

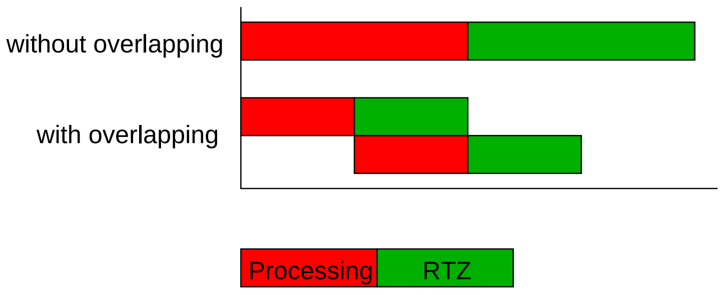


Figure 5.11: Overlapping processing and RTZ

be directed to the fetch unit instead of the register bank. This unit is also responsible for selecting the correct byte from the 32-bit word that is read from memory when a load byte instruction is used.

5.4.5 Pipelining issues

The issues related to pipelining in asynchronous systems are complex and a full discussion is beyond the scope of this thesis. Pipelining in this context, is used, not only to refer to the pipeline structure of the processor but also the additional fine-grained pipelining found within stages. This fine-grained pipelining is not present in synchronous designs but asynchronous designs usually feature it to some extent. A significant motivation for this pipelining is to reduce the overhead of the handshaking by the overlapping of the phases of the handshake.

Without overlapping, an entire pipeline stage must first go through the processing stage and then the return-to-zero phase. If the processing stage is split into two, then it is possible to overlap the return-to-zero of the first stage with the processing of the second stage. See figure 5.11 for an illustration of this idea. In order to perform this overlapping it is necessary to buffer the data between stages so that the data will not be made invalid by the RTZ phase of the first stage while the second stage is processing it. In the latest versions of Balsa, the overlapping and storage is implemented by the PassivatorPush component (see section 2.5.9). This is unfortunate as the placing of the PassivatorPush is not explicitly specified in the source description and so it is not straightforward to identify where the overlapping is being implemented. This makes it more challenging to translate to the data-driven style.

Additionally, the particular control used in the PassivatorPush is not available in the data-driven style, so even once the required location is identified there is no way of implementing exactly the same overlapping. In the data-driven style, the VariablePush has been used to implement the overlapping. This implements overlapping but it also goes further in that it does not just overlap processing with return-to-zero but fully decouples the two stages, allowing concurrent operation of the entire handshakes.

The structure of the fine-grained pipelining of the execute unit is therefore not precisely replicated in the data-driven style. This will have an effect on overall performance. The nature and extent of this is extremely difficult to analyse and beyond the scope of this thesis. It should be noted that no such analysis has been performed on the Balsa nanoSpa either and the pipelining structure has been developed by a mixture of intuition and trial-and-error. It is certainly not, therefore, the best possible solution for the original Balsa design. The optimal solution is specific to any given design, so even if the best possible solution for the original design were known it may not be the best for the data-driven design. It has been necessary to use some intuition and trial-and-error to attempt to replicate the original nanoSpa pipeline structure as closely as possible in the data-driven style.

An effect of this additional pipelining, related to the discussion in section 5.4.1, is that the maximum depth of branch shadows is increased. The additional pipelining means that more than one instruction may enter the execute unit at one time, allowing more instructions to be fetched following a branch than might be immediately expected. It is vital to exploit this asynchronous style of operation to improve performance but it does make reasoning about the operation of the pipeline a great deal more challenging than in synchronous design. At the time of writing, it is not actually known for certain how deep the maximum branch shadow is for either nanoSpa design! Providing more assistance for the designer in analysing and improving the pipelining and overlapping properties of a design is an interesting area for future work.

5.5 Simulation results

The results of simulations of the control-driven and data-driven nanoSpa processors are presented here. More instructive than simulating the entire processor is to examine the results for simulating individual modules from within the processor. This avoids issues associated with the pipelining and processor architecture and demonstrates the performance improvements gained by using the data-driven logic style. The fetch, decode and execute units have all been simulated. Additionally, some individual modules from the execute unit have also been simulated. These modules were simulated, where appropriate, with varying input data to demonstrate the data-dependent variation in performance. The environments used in the test benches for these simulations all have zero delay. Generally, this favours the control-driven approach as, for example, the cost of synchronising inputs that all arrive simultaneously is minimal. The results then, show (to a close approximation) the minimum improvement achieved by the data-driven style. In realistic operation, the fact that the data-driven style does not synchronise all inputs before beginning the operation and does not wait until the operation is complete before releasing them can potentially further improve performance.

All these simulations are performed at gate-level using fixed gate delays. This does not provide a highly accurate estimate of absolute performance although experience has shown that the results of these simulations closely approximate transistor-level simulations in a 180nm technology. As a relative measure for the comparison of the two styles this level of simulation is more than sufficient. The control-driven nanoSpa has previously been simulated at transistor-level and both bundled data and dual-rail implementations achieve approximately 55 Dhrystone MIPS. It can be seen from the results in the next sections that the gate-level simulations slightly under-estimate the transistor level performance.

Although the existing Balsa framework is being used for the back-end, some additional work is required to generate a transistor-level netlist that can be compared with equivalent simulations for the control-driven nanoSpa. This is because the example cell library used has been designed locally and only contains transistor level models for the precise cells needed to implement original Balsa components. Some data-driven components use cells that are not currently provided and these would need to be added to the cell library. This task has not been possible in time to provide results here. It is not believed that these results would yield any greater insight in the context of this thesis apart from giving a more accurate absolute performance estimate of the processor.

5.5.1 Dual-rail

The dual-rail control-driven nanoSpa achieves 50 Dhrystone MIPS. The data-driven version achieves 79 Dhrystone MIPS, an improvement of 1.6 times the original. The results for individual modules are shown in table 5.1. Area is compared in table 5.2.

It is clear that in general, the throughput of the data-driven modules is greater than the conventional Balsa equivalents. Interestingly, Balsa does well

Module	Test	Gates/cycle		Improvement
		Balsa	Data-driven	
Fetch		59	29	2.0
Decode	regular	52	39	1.3
	ldm/stm (5 registers)	604	254	2.4
Register bank	1 write	134	69	1.9
	2 writes	182	74	2.5
ALU	and	74	41	1.8
	add 0 carry	85	74	1.1
	add 5 carry	86	74	1.2
	add 32 carry	107	65	1.6
	sub 1 carry	87	74	1.2
	sub 5 carry	88	65	1.4
	sub 32 carry	109	65	1.7
	mov	77	57	1.4
ExecuteControl	non-memory	44	24	1.8
	memory store	57	30	1.9
	memory load	64	32	2.0
mux3		23	16	1.4
steerDi	load word	29	30	0.9
	load byte	53	54	0.9
Execute	nop	83	58	1.4
	and	93	58	1.6
	and with shift	133	65	2.0
	ands (update flags)	95	58	1.6
	ldr/str	116	65	1.8
	branch	92	74	1.2

Table 5.1: Dual-rail module results

Module	Balsa transistor count	DD transistor count
Fetch	7595	16757
Decode	62870	264200
Register bank	67036	370368
Execute		
ExecuteControl	4949	5578
ALU	38295	52296
Shifter	28503	82603
mux3	741	2590
steerDi	5951	10762
Other execute	47075	105635
Total execute	141846	259464
Other	36026	27880
Total	**315373**	**938669**

Table 5.2: Dual-rail nanoSpa area

with the steerDi module (see section 5.4.4) – the only module where the throughput is not improved in the data-driven version. It is not immediately clear why this should be the case. The nature of the module is unusual in that it takes a single input and selects which output to send it to. Speculation is not likely to have a positive effect in this case as there is really no data-processing to be performed. The data-driven control is large and as the operation is highly balanced, the use of T-elements to permit concurrent return-to-zero phases is probably not exploited and therefore the latency of the control has a significant impact on the throughput. However, further investigation is required to fully explain this result.

As expected the area is significantly increased. As anticipated, a significant proportion of this increase is found in the register bank. If the increase in register bank area is ignored, then the data-driven nanoSpa is roughly twice the size of the original Balsa version. The area overhead for dual-rail is particularly large. As noted in section 3.4 (page 73), this is mainly due to the large size of the VariablePush component and the increased number of completion

Module	Test	Gates/cycle Balsa	Gates/cycle Data-driven	Improvement
Fetch		47	33	1.4
Decode	regular	79	76	1.0
	ldm/stm (5 registers)	578	311	1.9
Register bank	1 write	82	61	1.3
	2 writes	99	61	1.6
ALU	and	57	33	1.7
	add 0 carry	75	42	1.8
	add 5 carry	83	52	1.6
	add 32 carry	137	106	1.3
	sub 1 carry	75	42	1.8
	sub 5 carry	85	52	1.6
	sub 32 carry	139	106	1.3
	mov	56	32	1.8
ExecuteControl	non-memory	41	32	1.3
	memory store	46	32	1.4
	memory load	50	32	1.6
mux3		18	14	1.3
steerDi	load word	24	22	1.1
	load byte	35	36	0.9
Execute	nop	70	56	1.3
	and	91	59	1.5
	and with shift	129	63	2.0
	ands (update flags)	91	59	1.5
	ldr/str	150	106	1.4
	branch	88	56	1.6

Table 5.3: Bundled data module results

detectors. Note that no attempt has been made to optimise the back-end component implementations for area so there is future potential for reducing the area overhead although given the magnitude of the performance gains, the area overhead is not excessive.

5.5.2 Bundled data

The bundled data control-driven nanoSpa achieves 52 Dhrystone MIPS. The data-driven version achieves 81 Dhrystone MIPS, an improvement of 1.5 times

Module	Balsa transistor count	DD transistor count
Fetch	4725	4837
Decode	36382	58471
Register bank	30480	79480
Execute		
ExecuteControl	2779	3358
ALU	9079	11142
Shifter	13405	25105
mux3	919	926
steerDi	1807	2833
Other execute	27813	27436
Total execute	55802	70800
Other	20172	9795
Total	**147561**	**223383**

Table 5.4: Bundled data nanoSpa area

the original. The results for individual modules are shown in table 5.3. Area is compared in table 5.4.

The improvements in throughput of the individual modules are fairly similar to those shown in the dual-rail implementation. Again steerDi is the exception to the general trend which will be due to the same reason as in dual-rail. The decode of regular single-cycle instructions is also not improved significantly and this is explained in the section on the decode unit above. An interesting property can be seen in the ExecuteControl unit where the impact of speculation can be clearly observed in the data-driven results which are equal for all three different operations.

The difference in area for the bundled data implementation is much smaller than that for dual-rail. Again, much of the increase is in the register bank. If the increase in register bank area is ignored then the data-driven design is only approximately 18% larger. When compared with the magnitude of the performance improvement, this area overhead can be considered as insignificant.

5.6 Register bank hybrid design

The register bank has been highlighted as a particular problem in terms of area and energy consumption. A possible solution that may be easily implemented is to use the conventional Balsa register bank in place of the data-driven register bank. As the interface to both register bank designs is the same and the two design styles are integrated into the same flow, it is trivial to produce this hybrid design.

This provides an excellent example of how designs with mixed Balsa and data-driven modules can be used. The lower area and energy requirements of the control-driven style can be exploited for non-critical modules, while the performance of the data-driven style is exploited for others.

The new hybrid design achieves 62 MIPS in dual-rail and uses 637,119 transistors. In bundled data the performance is 67 MIPS and the size is 175,635 transistors. Performance has been traded for reductions in area and energy consumption.

Chapter 6

Conclusions

This thesis has described contributions in the field of asynchronous digital circuit synthesis. The existing Balsa synthesis method has been examined and performance has been identified as a major weakness. The overhead of the control-driven style of compilation has been identified as a significant contributing factor to the shortcomings in performance of the existing synthesis method. However, the handshake circuit paradigm is attractive because it is both flexible and robust, independent of any particular implementation style, straightforward to understand, and the transparent compilation allows source-level optimisation.

A data-driven style of circuit would seem to offer potential for increased performance. Therefore an alternative data-driven style of handshake circuit structure has been proposed along with a language from which this circuit style may be compiled. The compiler to translate this language into handshake circuits has been implemented and integrated into the existing Balsa framework.

The data-driven style has been successfully demonstrated by the implementation of a complex 32-bit microprocessor design. The potential performance improvements over the control-driven style have been convincingly demonstrated by comparison of this design with the equivalent control-driven implementation.

The increased area and energy requirements of the data-driven style have been briefly noted but these are unlikely to be disproportionate to the performance gains and could be decreased by further work on modified or alternative back-end implementation styles.

A drawback of the data-driven style is that the descriptions are arguably less flexible and not as familiar to a general hardware designer as those that are possible in Balsa. It is difficult to draw conclusions about the reaction of any given designer when encountering a particular tool and language for the first time as such judgements are highly subjective. The experience of translating the nanoSpa design would indicate that, at least for Balsa code written for performance, the reduced flexibility is not a significant impediment with the one particular exception of the register bank. Of course, the Balsa nanoSpa was already written in a fashion that was essentially data-driven anyway.

Due to the variables and sequential and iterative control structures, it is possible in Balsa to write a naive sequential program that appears very similar to a conventional programming language. Such a program will compile and produce a functioning (but slow) circuit. In the data-driven style, it is arguably necessary for the programmer to think in a different, more 'asynchronous' manner as such sequential descriptions are not possible. It is also similarly necessary to do so when using conventional Balsa if good performance is required. The rewards of adopting a data-driven style with respect to performance are clear but the method introduced herein, being intentionally designed to be data-driven, is clearly superior to adopting a data-driven approach with control-driven compilation. By using the handshake circuit paradigm, it is straightforward to combine both styles in the same design-flow and so greater flexibility is offered to the designer. For example, a possible scenario is that, in a large system, some critical parts of a design might be implemented data-driven by a more experienced designer, while less critical parts are implemented control-driven by a less experienced designer.

The data-driven style has addressed the issue of the structure of handshake circuits and control overhead. This is a very useful contribution but is by no means the end of the story. In general, the performance of synthesised asynchronous circuits is still not competitive with their synchronous counterparts. More work is required at all levels of the design-flow before competitive performance is achieved. The next section will discuss some ideas for future work arising immediately from the data-driven handshake circuit style and the nanoSpa example design.

6.1 Future work

The possibilities for work extending, using and based upon the data-driven circuit style and language are innumerable. During the course of the last two chapters, a number of areas for future work have been identified. This section collects these ideas and adds a few additional ones.

6.1.1 Language and compiler

There is much potential for developing and extending the data-driven language and compiler. A few ideas are briefly presented here.

More concise syntax

Sometimes the data-driven language is quite verbose. For example it is not uncommon to write the following block:

```
input  a
output b
during
    b <- a
end
```

Syntactic shortcuts for some code patterns could be added to the language. For example, `a -> b`, could be offered as a quick shortcut for the above block.

Iteration

Conventional Balsa contains the while construct to support temporal iteration. The data-driven language contains no explicit temporal iterative structure. The control-driven style allows for this control structure to be implemented easily but it is very poor for performance as demonstrated by the nanoSpa decode unit when decoding multi-cycle instructions (see section 5.4.2 on page 139 and tables 5.1 and 5.3 on pages 152 and 154). The data-flow style implementation of the data-driven nanoSpa decode unit is a much more efficient implementation of iteration. It would seem likely that this type of structure will recur and so an idea for future work is to add some convenient construct to the language that will allow it to be generated easily. Ideally this could be done in a manner that closely resembled the while loop structure of Balsa and other common languages. The structure used in the decode unit is probably obvious to an experienced asynchronous designer but a less experienced designer would probably benefit from a more familiar control-type language construct that is compiled into the efficient data flow structure.

Fine-grained integration

In section 4.1 (page 115), the possibility of tighter integration between the two design styles was discussed. The nanoSpa design has highlighted areas where such integration or perhaps even more fine-grained integration might prove useful. The register bank is a clear problem in the data-driven style. Not only is the area requirement significantly larger but the energy required to write every register in every cycle probably makes this design infeasible. Balsa style variables are clearly more suited to the register bank application but the control-driven logic means the performance of the Balsa register bank is limited. Using the Variable component in the data-driven style is difficult as the explicit sequencing control structures that prevent concurrent reads and writes to the variable are not available.

Is there some way in which Variable components could be used with data-driven style logic with the addition of some control-driven elements to ensure that concurrent read and write hazards do not occur at the Variable? How might this be expressed in the language?

Automated translation

The idea of automatically translating Balsa handshake circuits into the data-driven style was discussed quite extensively in section 4.4 (page 126). This is a particularly interesting idea as it would allow some of the performance benefits of the data-driven style to be exploited by conventional Balsa descriptions written using the more familiar sequential and iterative structures found therein.

6.1.2 Back-end and components

There is scope for optimisations at the back-end stage of compilation and in component implementations. These may be targeted at performance, area, power or a combination of all three.

Control re-synthesis

The technique of control re-synthesis was discussed in section 3.1.1 (page 67) and this could be applied to the control sections of data-driven circuits just as easily as to control trees.

Improved components

In section 4.2 (page 117), the possibility that some component implementations may implement too much concurrency for certain circumstances was mentioned. This may have be demonstrated by the steerDi module from the nanoSpa execute unit (section 5.4.4) although this requires further investigation. The possibility of using modified components in certain circumstances is worthy of future investigation. This may be possible by so-called 'peephole' style optimisation where a pattern of components is identified and replaced with an alternative. It may also require more complex analysis of the behaviour of the circuit.

Reduce area overhead

The nanoSpa example has shown that the data-driven style generally uses more area than conventional Balsa circuits. This is particularly true in delay-insensitive implementations. Reducing this overhead is a possible area of future work. For example, it might be possible to reduce the number of completion detectors required by re-using other completion detectors in some circumstances. The data-driven style adds a completion detector at every point where data may be rejected, but there is also another completion detector for the same data later in the datapath for the case where it is not rejected. Is it possible to combine these in some fashion to save area?

Combining with other techniques

It has been previously noted that most asynchronous design techniques are in general data-driven rather than control-driven. Many asynchronous design styles are based on the pipeline approach of Sutherland's Micropipelines [Sut89]. High-performance asynchronous pipeline implementations is an active area of research[SN01, YBA96, AML97]. The data-driven handshake circuit style is also essentially a pipeline approach. Therefore the possibility exists to implement back-ends to the data-driven style using other high-performance pipeline style techniques.

Other research in the asynchronous field is also more suited for use with the data-driven style and could be combined with it to further improve performance. For example, early output logic [Bre06] could be combined very easily with the dual-rail back end and should help to further improve performance.

6.1.3 nanoSpa

Although the nanoSpa processor itself is not the focus of this thesis, a couple of issues have been touched upon in this context that expose general issues for future work.

Register banks

In section 5.4.3 (page 142), the requirements for the register bank of nanoSpa were discussed. It was noted that an architecture with a longer pipeline than three stages would require a register bank where reads and writes can occur concurrently, and a mechanism to prevent hazards. An area of future work lies in finding some suitable method of synthesising this register bank, either with the presently available features of the synthesis system, or more likely with extensions to the synthesis system.

Pipeline analysis and optimisation

Section 5.4.5 (page 148) highlighted the issue of pipelining in asynchronous designs. Pipelining is an important factor in the performance of asynchronous designs. Clearly, a more systematic approach to analysis and optimisation than those presently used in Balsa or data-driven synthesis would be very useful. Tool support for such analysis and optimisation would greatly assist the designer. There are existing approaches to this problem including 'Slack Matching' [PM06] and 'Blame Passing' [Bre06]. An opportunity for future work is to examine the possibility of integrating these or other techniques into the synthesis flow.

6.2 Summary

A data-driven style of circuit clearly suffers from less control overhead than the control-driven style of Balsa. This thesis has shown that a data-driven style can be compiled using the handshake circuit paradigm in a syntax-directed fashion. It has been necessary to reduce somewhat the flexibility of the source description language compared to the conventional Balsa language. However, it has been demonstrated that the new handshake circuit style can be used to construct a significant design example. Results show that the new handshake circuit structure produces circuits with significantly increased throughput over those of Balsa, even in conditions favourable to the Balsa circuits.

The data-driven style is not an end in itself but hopefully offers a useful contribution towards the main goal of producing an asynchronous synthesis method that can construct large-scale designs and that will offer performance competitive with synchronous methods.

Appendix A

Language Grammar

The grammar of the data-driven language is presented here in an extended BNF form. The following conventions are used:

- Terminals are denoted by **bold** type.
- (a)* denotes zero or more repetitions of a.
- (a)+ denotes one or more repetitions of a.
- (a)? indicates that a is optional.

file	=	(outer)* **eof**
outer	=	((proc) | (type_dec) | (constant))
type_dec	=	**type id is** ((type) | (**record** (**id :** type (**;** | **,**)?)+ ((**end**) | (**over** type))) | (**enumeration** (**id** (**=** expression)?) (**,** **id** (**=** expression)?)* ((**over** type) | (**end**))))
constant	=	**constant id =** expression (**:** type)?
proc	=	**procedure id** ((**(** proc_spec **)** **is** declarations **begin** body **end**) | (**is** call))

proc_spec	=	proc_param_spec proc_ports_spec
proc_param_spec	=	((**parameter id :** ((type) | (**type**)) (**;** | **,**)?))*
proc_ports_spec	=	(((**array** range **of**)? (((**input** | **output**) **id :** type) | (**sync id**)) (**;** | **,**)?))*
declarations	=	((declaration) | (type_dec) | (constant))*
declaration	=	((**array** range **of**)? ((**channel** identifiers **:** type) | (**sync** identifiers) | (**variable** identifiers **:** type)))
body	=	(((block) | (init) | (call) | (iterative_call)))*
init	=	**init** block_body **end**
block	=	(**input** subject_list)? (**output** subject_list)? ((**during** block_body **end**) | (**always** always_body **end**))
call	=	**id (** proc_actuals **)** (**(** (((channel_spec) | ({ (channel_spec (**,**)?)* })) (**,**)?)* **)**)?
iterative_call	=	**for id in** range (**then**)? (call)+ **end**
proc_actuals	=	(((expression ((**bits**) | (**signed bits**))?) | (**array** range **of** type))) (**,** (((expression ((**bits**) | (**signed bits**))?) | (**array** range **of** type))))*
channel_spec	=	**id** (**[** range **]**)?
subject_list	=	channel_spec (**,** channel_spec)*
block_body	=	((case | if | write | for | foreach | arbitrate | (**continue**)))+
always_body	=	((case | if | write | for | foreach | arbitrate | sync | (**continue**)))+
write	=	(**all**)? channel_spec **<-** expression
case	=	**case** expression **of** expressions **then** inbody (| expressions **then** inbody)* (**else** inbody)? **end**
if	=	**if** expression **then** inbody (| expression **then** inbody)* (**else** inbody)? **end**

arbitrate	=	**arbitrate** subject_list **then** inbody | subject_list **then** inbody **end**
inbody	=	((case | if | write | for | foreach | arbitrate | sync | subject_in | (**continue**)))+
subject_in	=	**input** subject_list **during** block_body **end**
sync	=	**sync** channel_spec ((**;**) **sync** channel_spec)*
for	=	**for id in** range (**then**)? block_body **end**
foreach	=	**foreach id in** channel_spec (**then**)? block_body **end**
type	=	((expression ((**signed**)? **bits**)?) | (**array** range **of** type))
identifiers	=	**id** (**,** **id**)*
range	=	((expression (**..** expression)?) | (**over** type))
expressions	=	expression (**,** expression)*
expression	=	bitwise_exp
bitwise_exp	=	and_exp ((**or** | **xor**) and_exp)*
and_exp	=	equal_exp (**and** equal_exp)*
equal_exp	=	comp_exp ((**=** | (**/=** | <>)) comp_exp)*
comp_exp	=	concatenation ((<**=** | >**=** | < | >) concatenation)*
concatenation	=	numeric_exp (**@** numeric_exp)*
numeric_exp	=	term ((**+** | **-**) term)*
term	=	unary_exp ((***** | / | **%**) unary_exp)*
unary_exp	=	(unary_operator)? sizeof
sizeof	=	(**sizeof**)? exponentiation
exponentiation	=	array_index (ˆ array_index)*

array_index = smashed ((**[** range **]**) (**.** **id**)*)*

smashed = (**#**)? record_access

record_access = primitive (**.** **id**)*

func_args = ((expression ((**signed**)? **bits**)?)
| (**array** range **of** type))
((**,**) ((expression ((**signed**)? **bits**)?)
| (**array** range **of** type)))*

primitive = ((**id** ((**'** **id**) | (**expressions**) |
(**(** (func_args)? **)**))?) | literal | **string** |
(**(** bitwise_exp (**as** type)? **)**) | ({ expressions }))

unary_operator = (**-** | **not** | **log**)

literal = (**int** | **octal** | **hex** | **binary**)

Appendix B

New Handshake Components

This appendix contains details of the implementations of the new handshake components used in the data-driven style. Push equivalents for original datapath components are not shown as these implementations differ from the original pull versions only in the reversing of the protocol. For each component an STG is given to describe the behaviour and circuits are shown for both bundled-data and dual-rail implementations.

In the case of bundled-data implementations, usually only the control part of the circuit is shown. Where this is so, it can be assumed that the data path is simply wires from input to output, forking if necessary to feed multiple outputs.

Parameterisable component implementations are shown expanded with a particular specification which is given in each case. Hopefully, it is clear how to extend each implementation to implement any specification of the parameters.

Note that no claim is made that these implementations are the most suitable or efficient in terms of performance, area or power, although a desire for performance was considered during the design process. The principal motivation was to produce designs that would enable working circuits to be synthesised and simulated. It is intended that the dual-rail and bundled-data control circuits are quasi delay insensitive but this has not been formally verified.

B.1 VariablePush

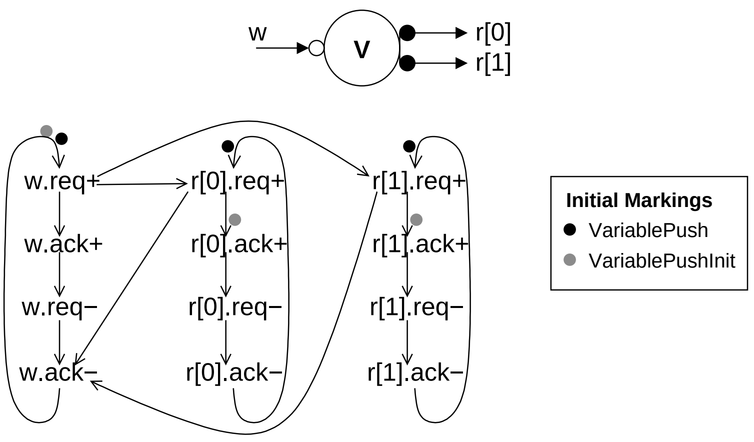


Figure B.1: VariablePush STG

VariablePush is a component of significant complexity. The STG in figure B.1 gives the STG for the external behaviour in the dual-rail implementation with two read ports. Two different initial markings are given; one for standard VariablePush and one for the initialised version.

The single-rail implementation (figure B.2) is based on a fully-decoupled four-phase micropipeline-style latch controller [Liu97].

The dual-rail implementation (figure B.3) is based on the standard Muller Pipeline [Mul62] and internally uses bit-level completion detection to avoid large C-element trees to perform completion detection across the entire width of the data. The circuit shown is for a width of two bits and two read ports. When parameters of a greater magnitude are used the figure soon becomes very large.

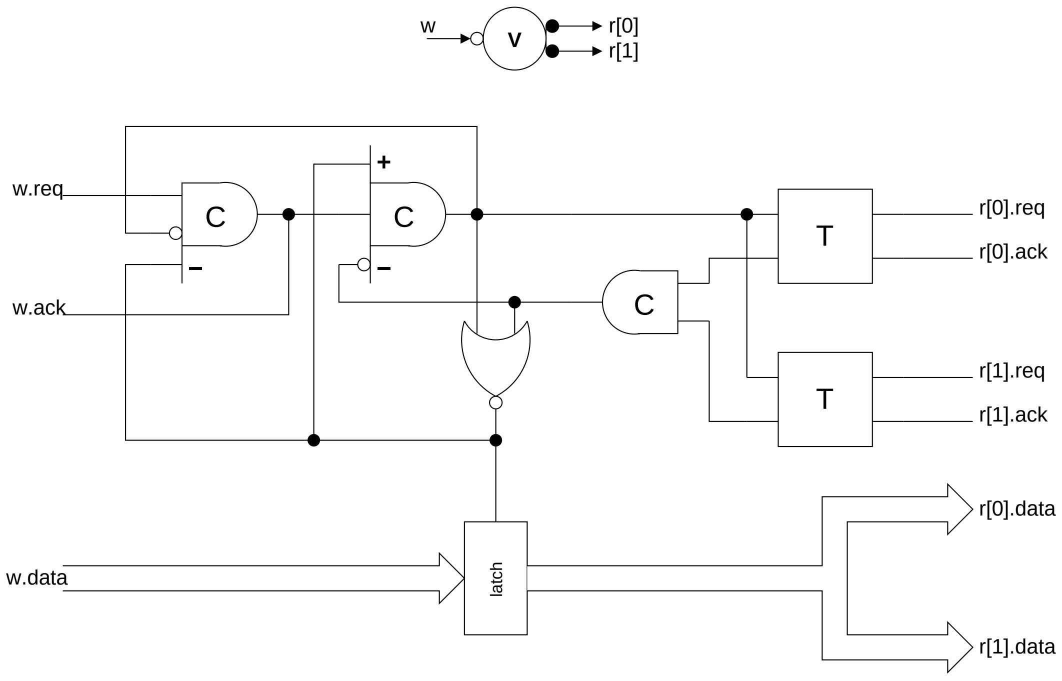


Figure B.2: VariablePush bundled data circuit

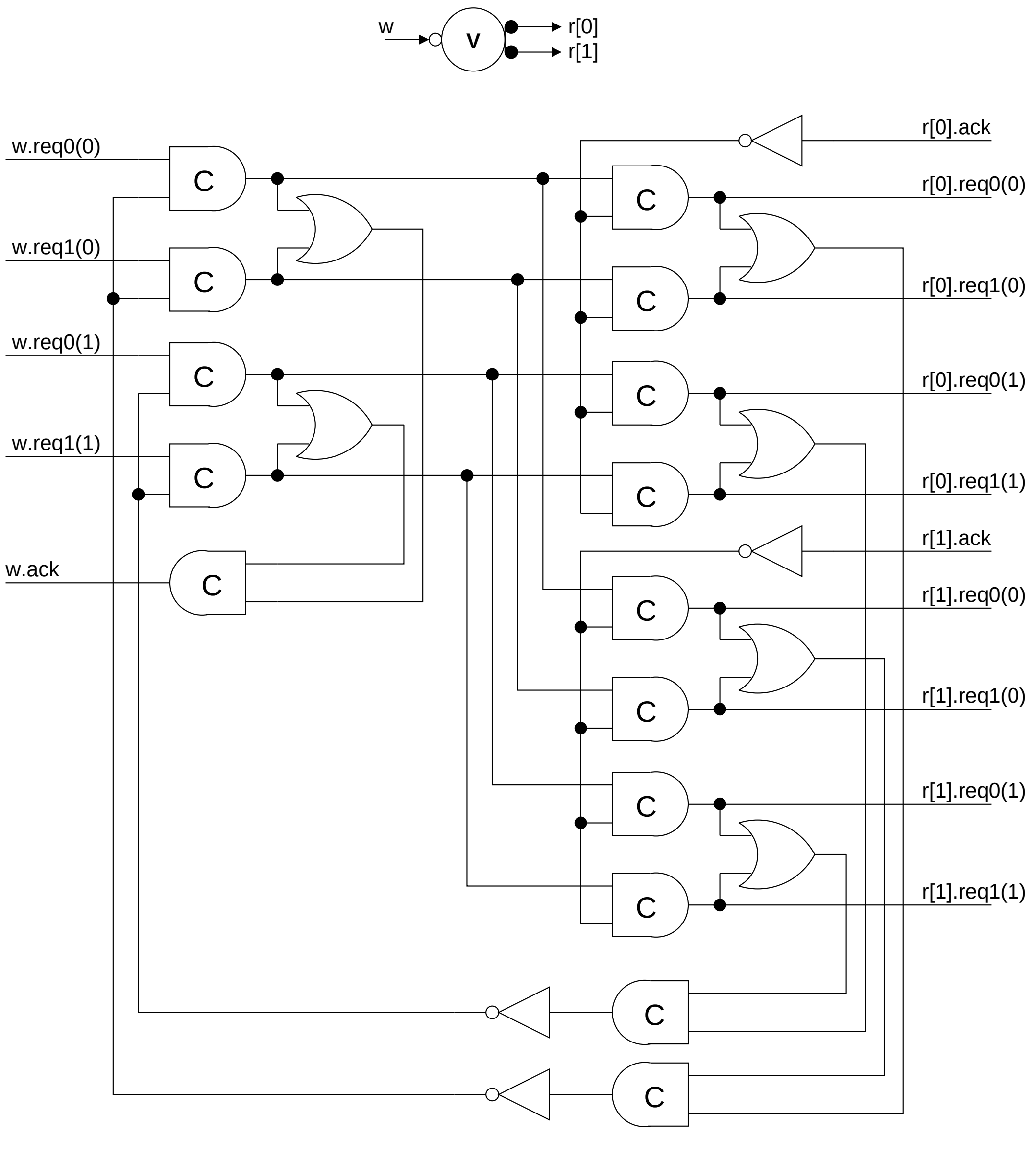


Figure B.3: VariablePush dual-rail circuit

B.2 Dup

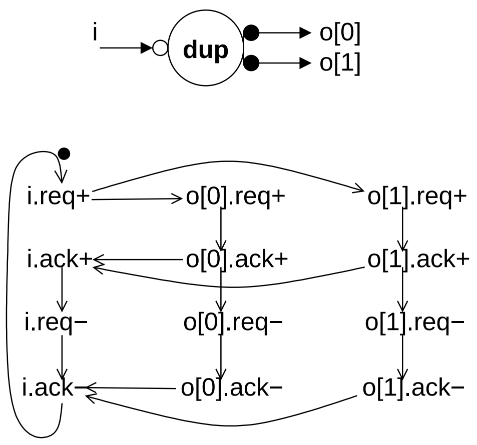


Figure B.4: Dup STG

Dup is used to ‘duplicate’ the communication on one input channel to multiple output channels. The term duplicate is used to differentiate the operation from a fork. Dup allows the handshakes on each output channel to operate independently. In a four-phase implementation, a fork would typically force synchronisation between the processing and return-to-zero phases of all handshakes.

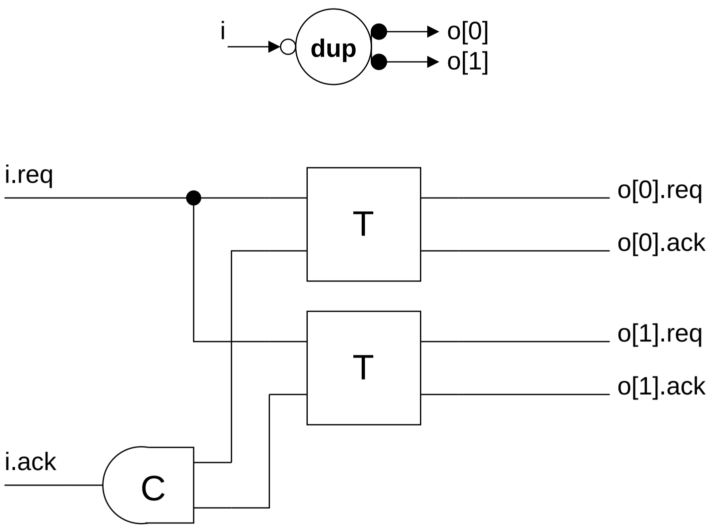


Figure B.5: Dup bundled-data control circuit

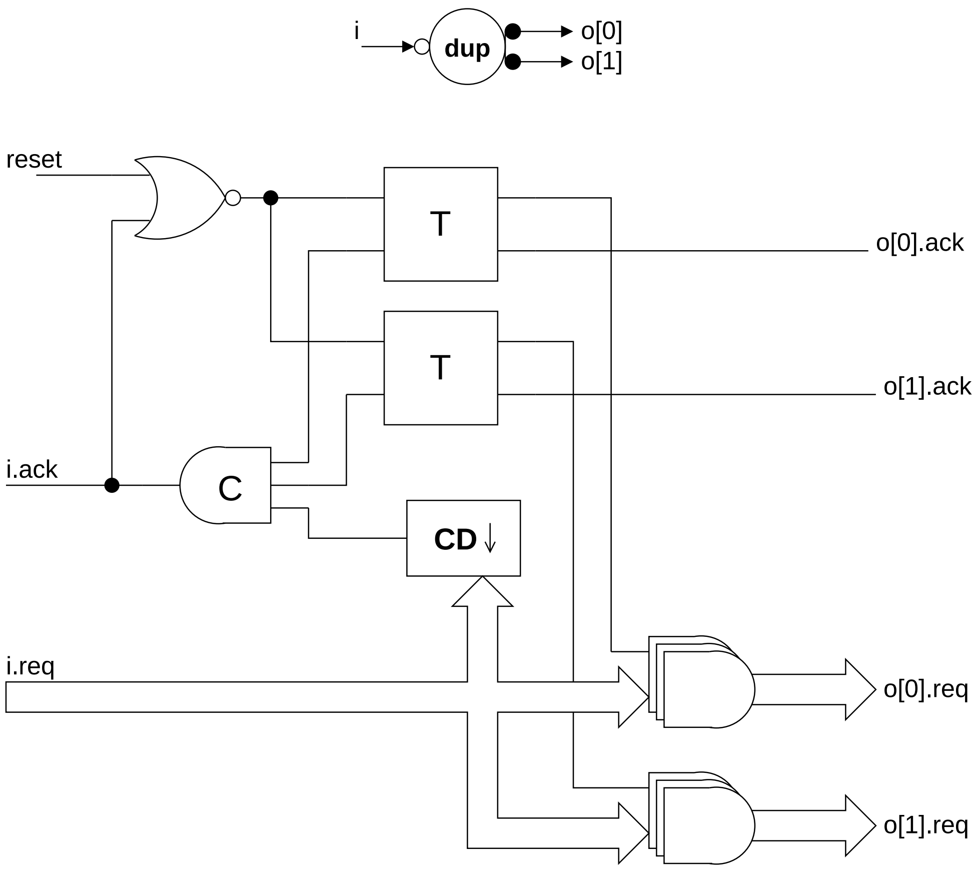


Figure B.6: Dup dual-rail circuit

B.3 FVPush

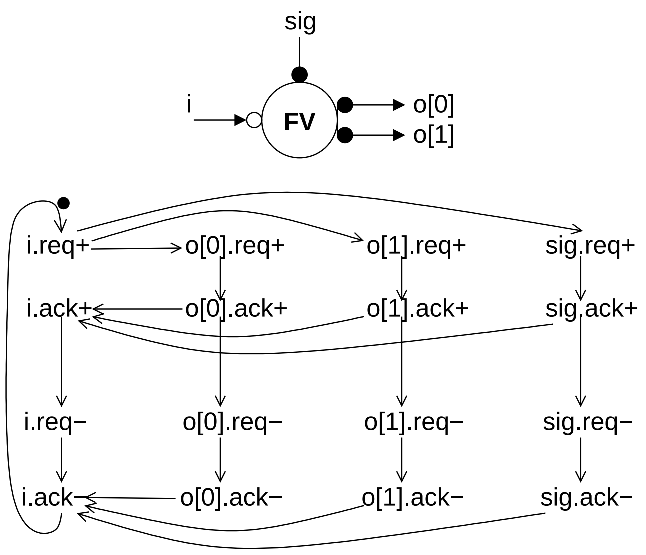


Figure B.7: FVPush STG

FVPush is essentially the same as a Dup with the addition of a sync signal port that is used to indicate arrival of the input data.

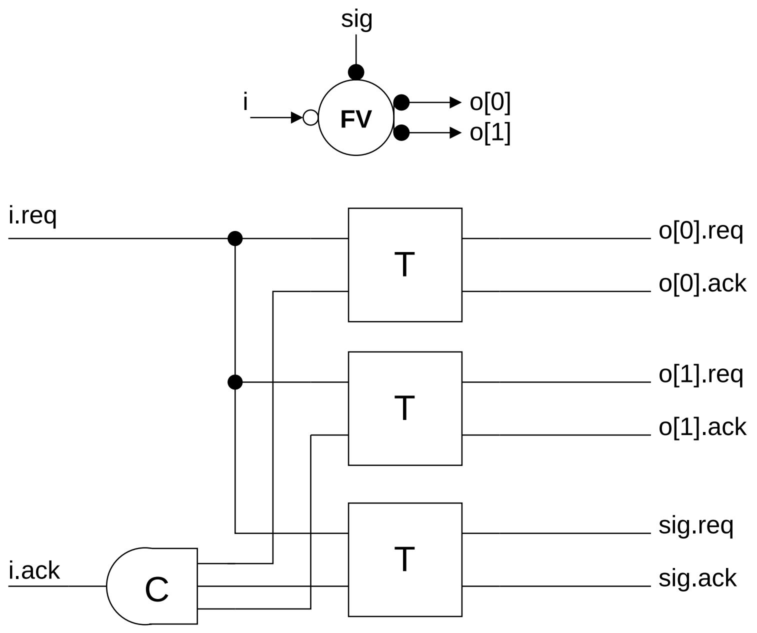


Figure B.8: FVPush bundled-data control circuit

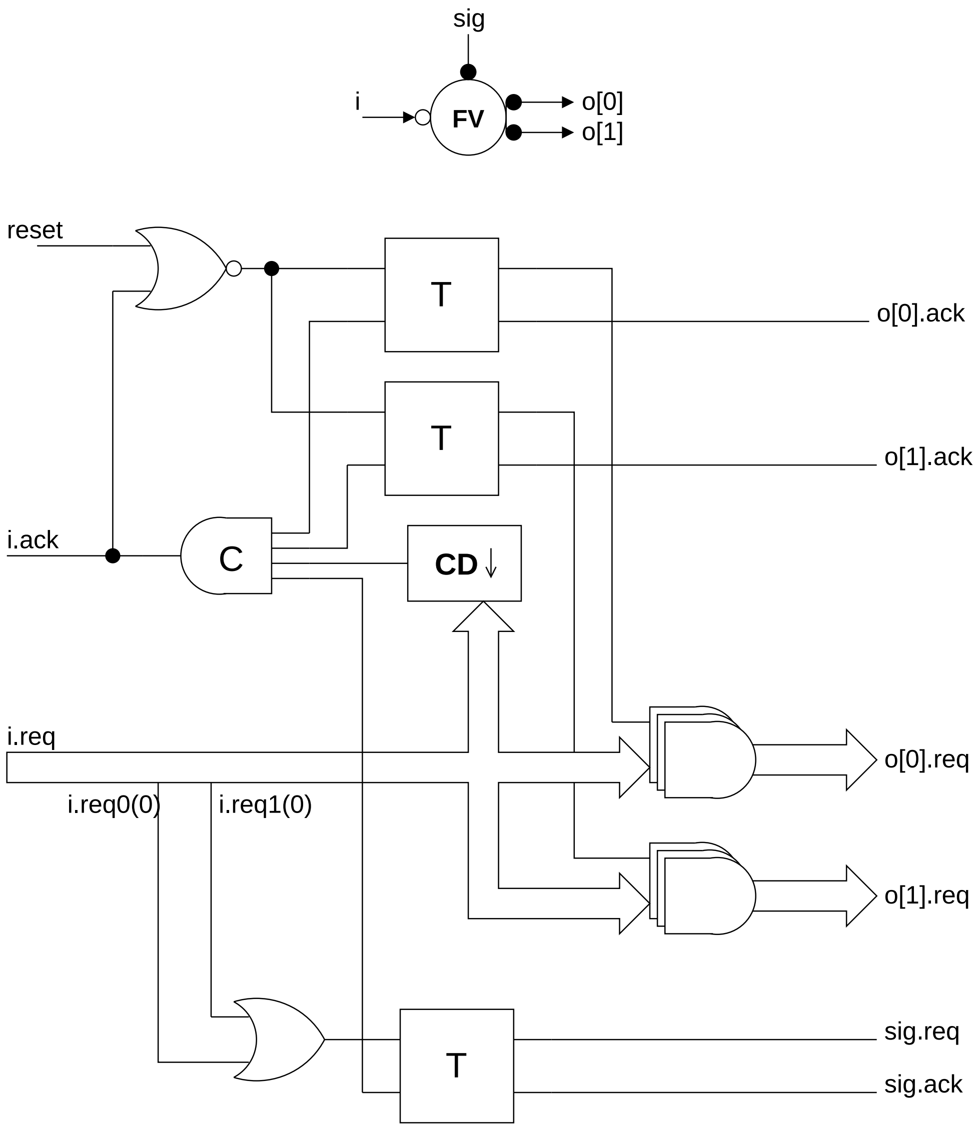


Figure B.9: FVPush dual-rail circuit

B.4 FetchPush

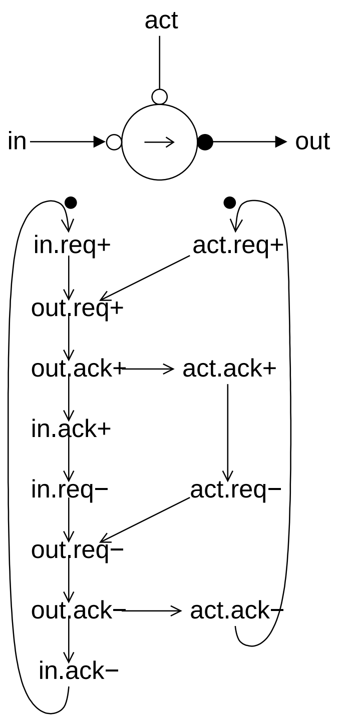


Figure B.10: FetchPush STG

FetchPush is used to synchronise data and control where it is not possible for data to continue without indication from the control that it is safe to do so. The data arriving on the input port is synchronised with the activation before being passed to the output port.

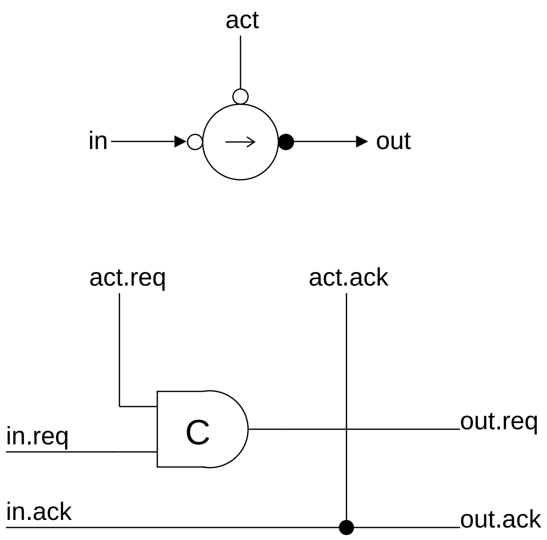


Figure B.11: FetchPush bundled-data control circuit

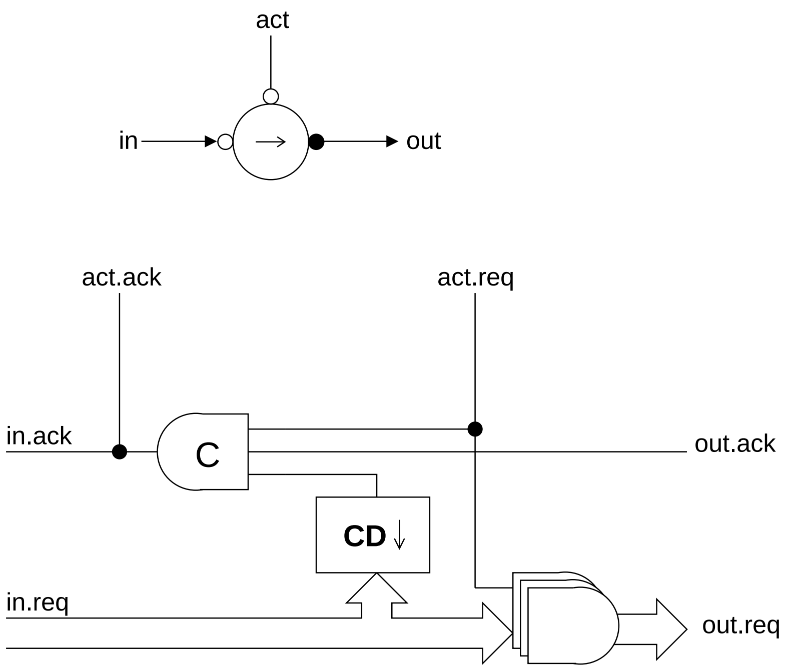


Figure B.12: FetchPush dual-rail circuit

B.5 FetchReject

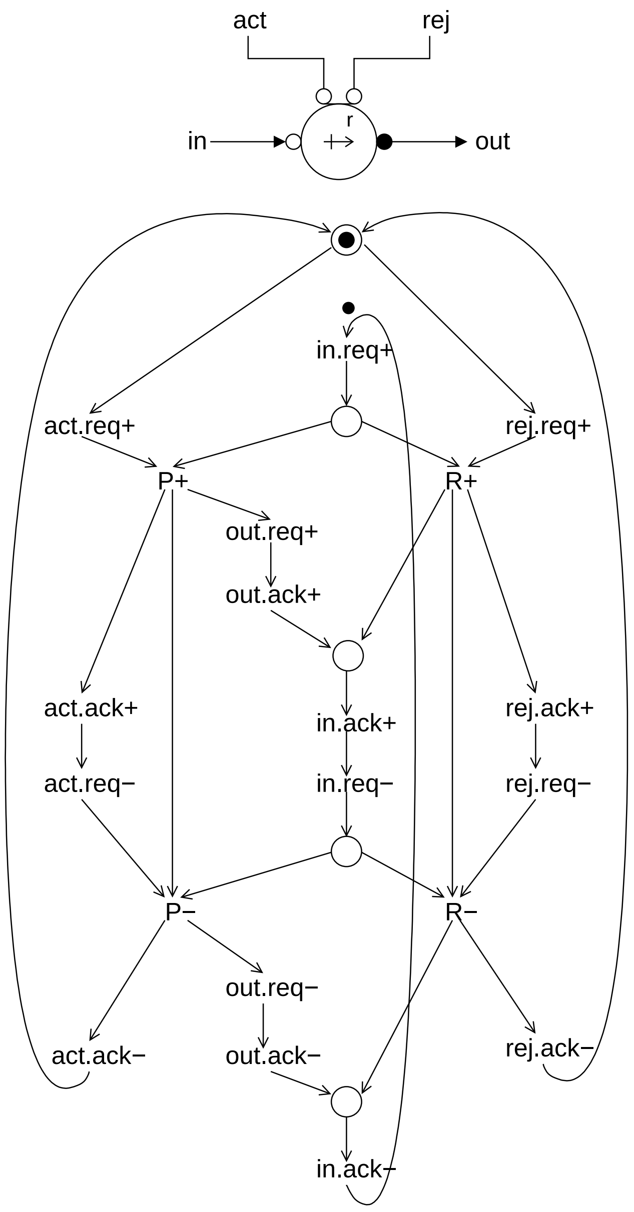


Figure B.13: FetchReject STG

FetchReject is similar to FetchPush with the addition of a ‘reject’ port that allows the input data to be discarded instead of passed. Handshakes on the activation and reject ports must be mutually exclusive.

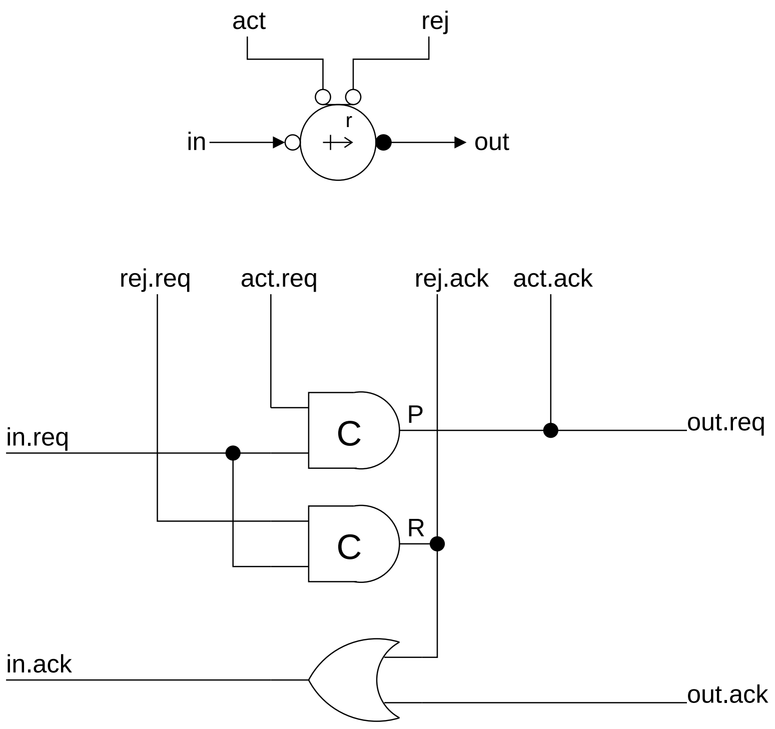


Figure B.14: FetchReject bundled-data control circuit

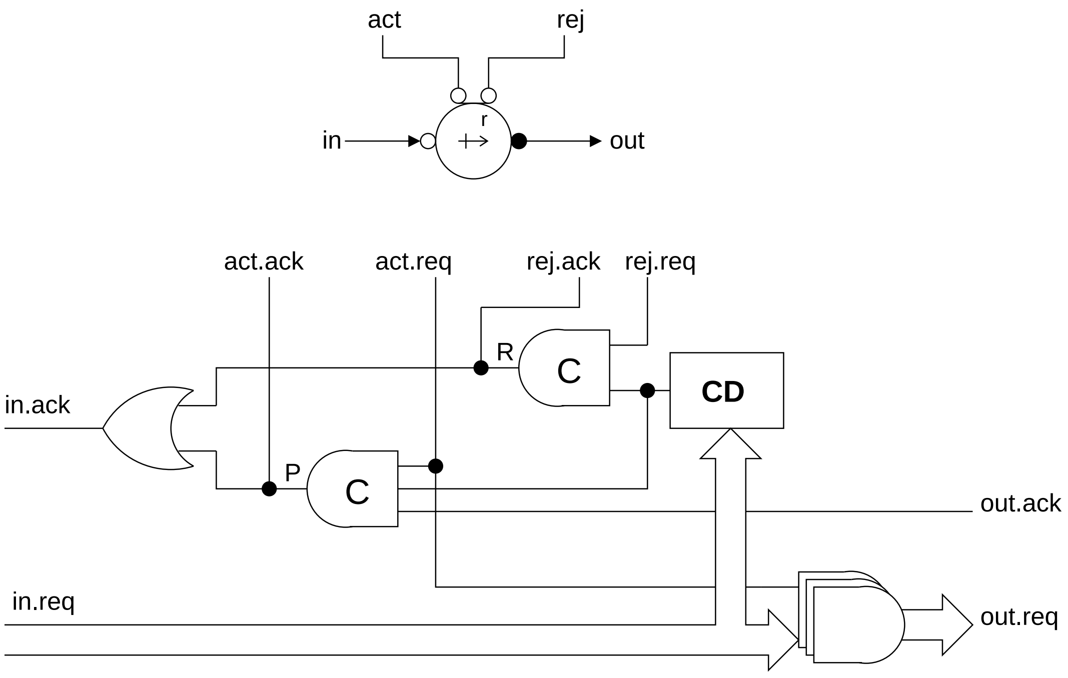


Figure B.15: FetchReject dual-rail circuit

B.6 CasePush

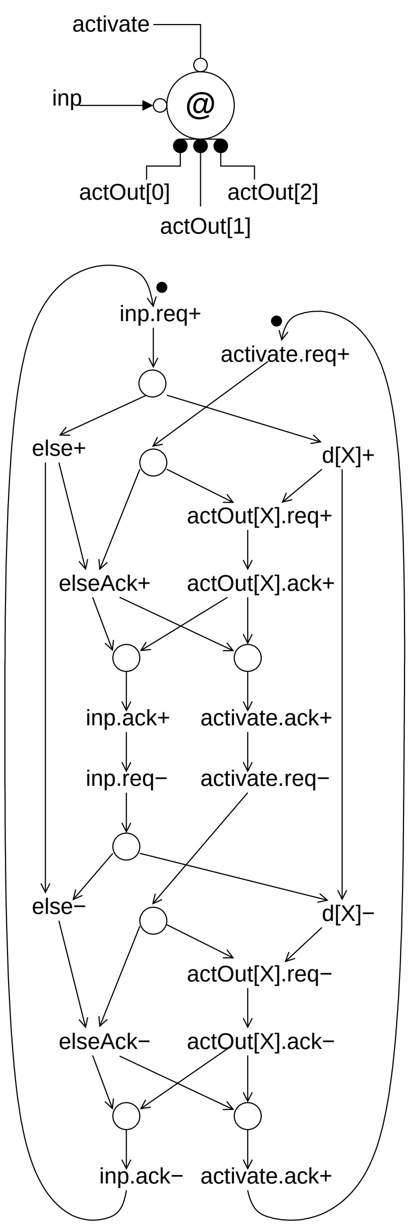


Figure B.16: CasePush STG

CasePush is used to implement nested conditional structures. Only one output activation will be produced for each input that arrives. The output activation will not be produced until the control indicates it is safe to do so by activating the component.

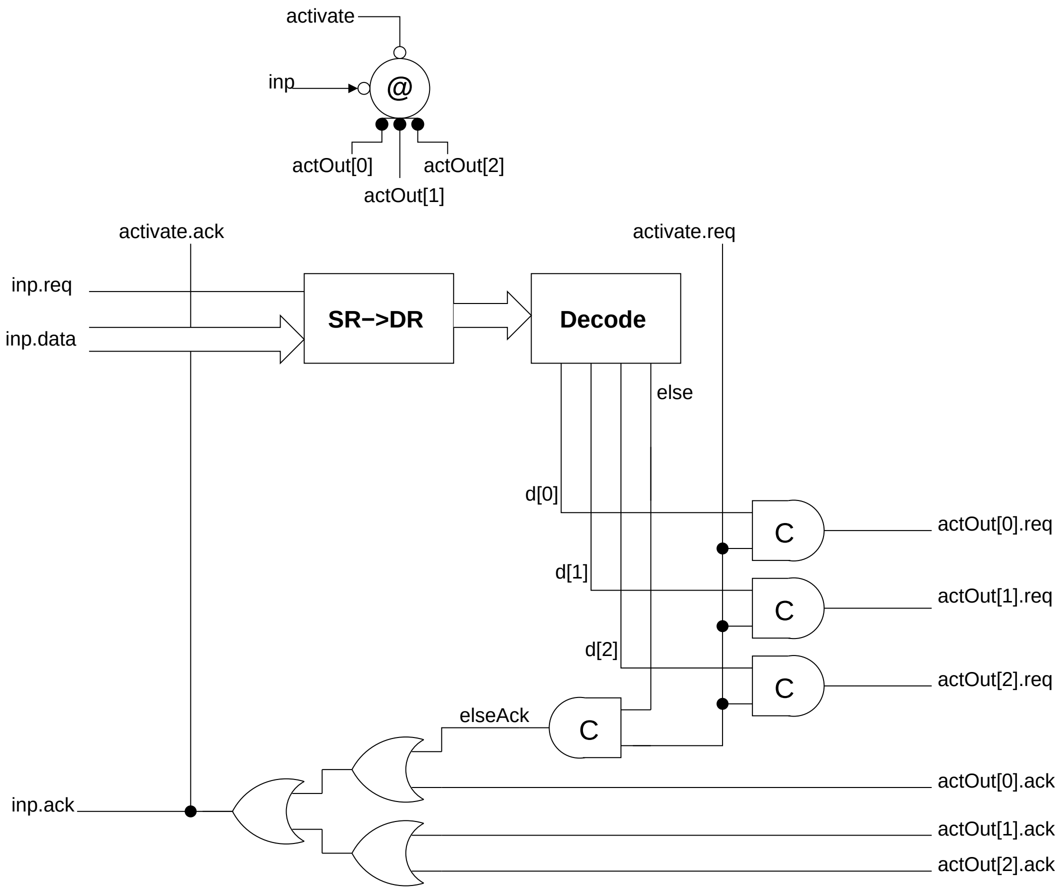


Figure B.17: CasePush bundled-data circuit

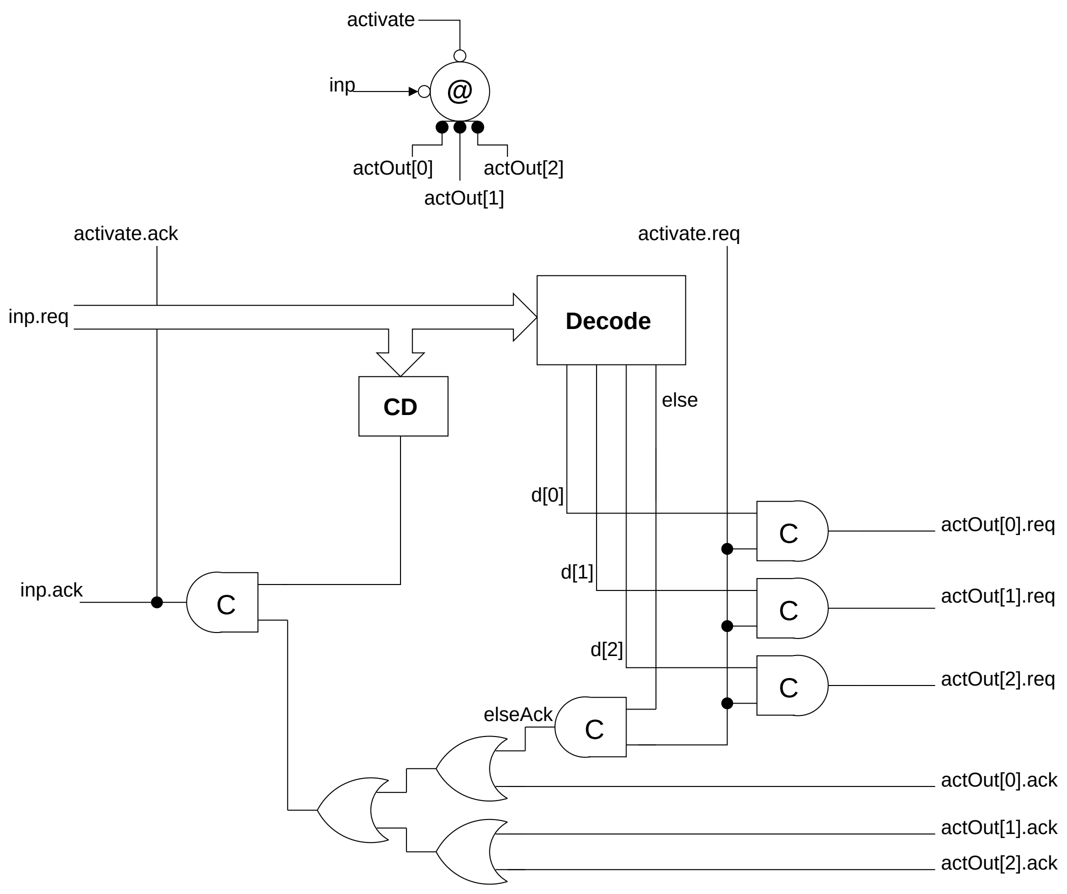
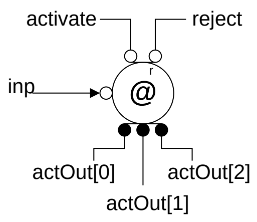


Figure B.18: CasePush dual-rail circuit

B.7 CasePushR



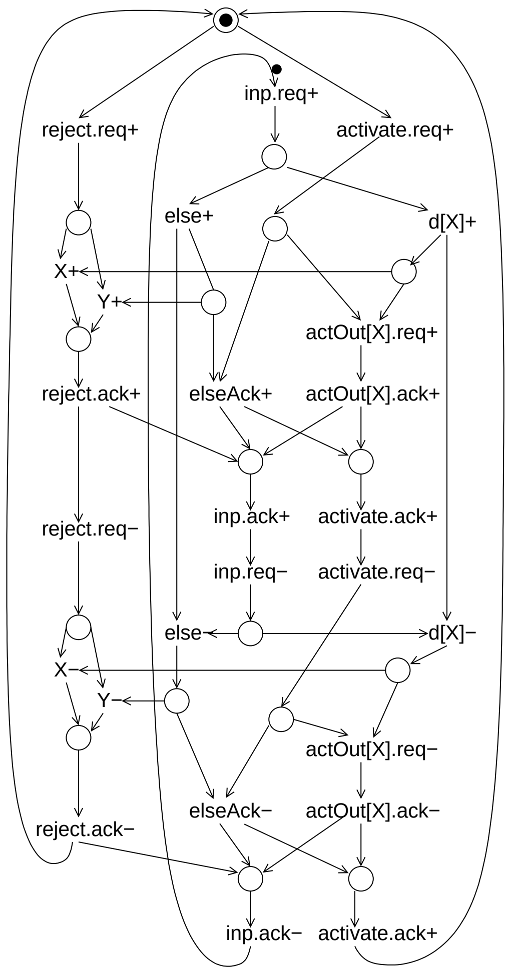


Figure B.19: CasePushR STG

CasePushR is very similar to CasePush. In addition to the activation, a reject port is provided, allowing the control to discard the input data without activating an output. Handshakes on the activate and reject ports must be mutually exclusive.

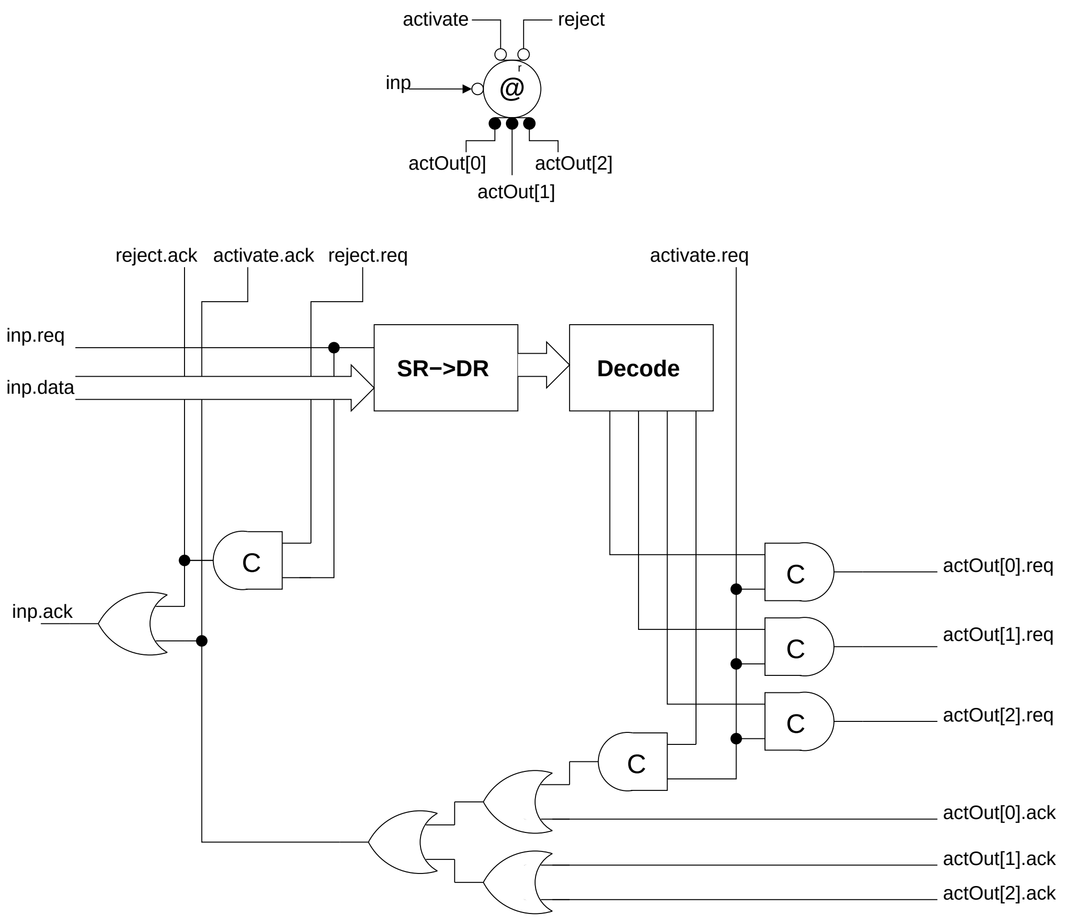


Figure B.20: CasePushR bundled-data circuit

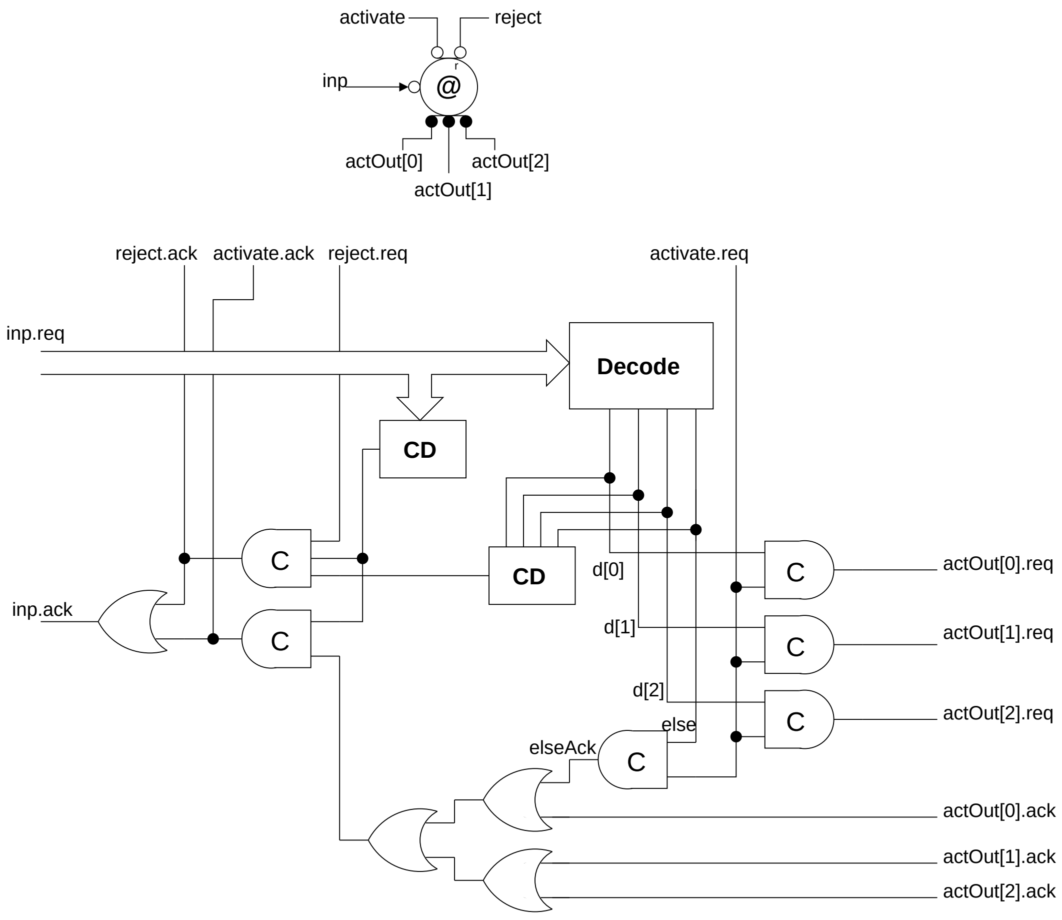
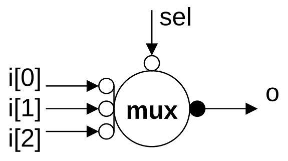


Figure B.21: CasePushR dual-rail circuit

B.8 Mux



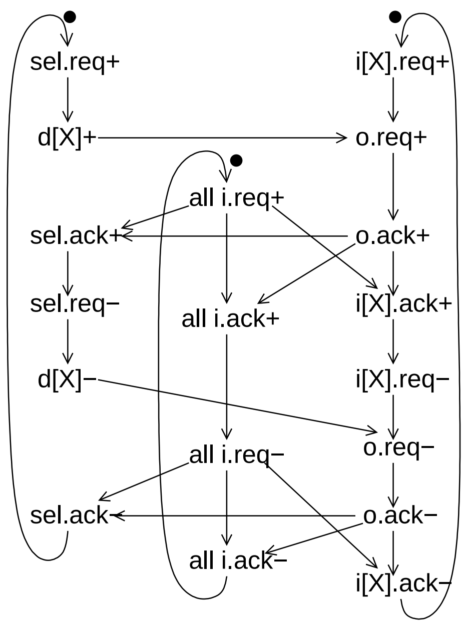


Figure B.22: Mux STG

Mux is used to implement run-time array indexing. It will pass one of its input ports to the output based on the value of the sel(ection) input. The data on all other input ports is discarded. The implementations have been designed to output the required data early, in that not all the inputs must arrive before the output is produced.

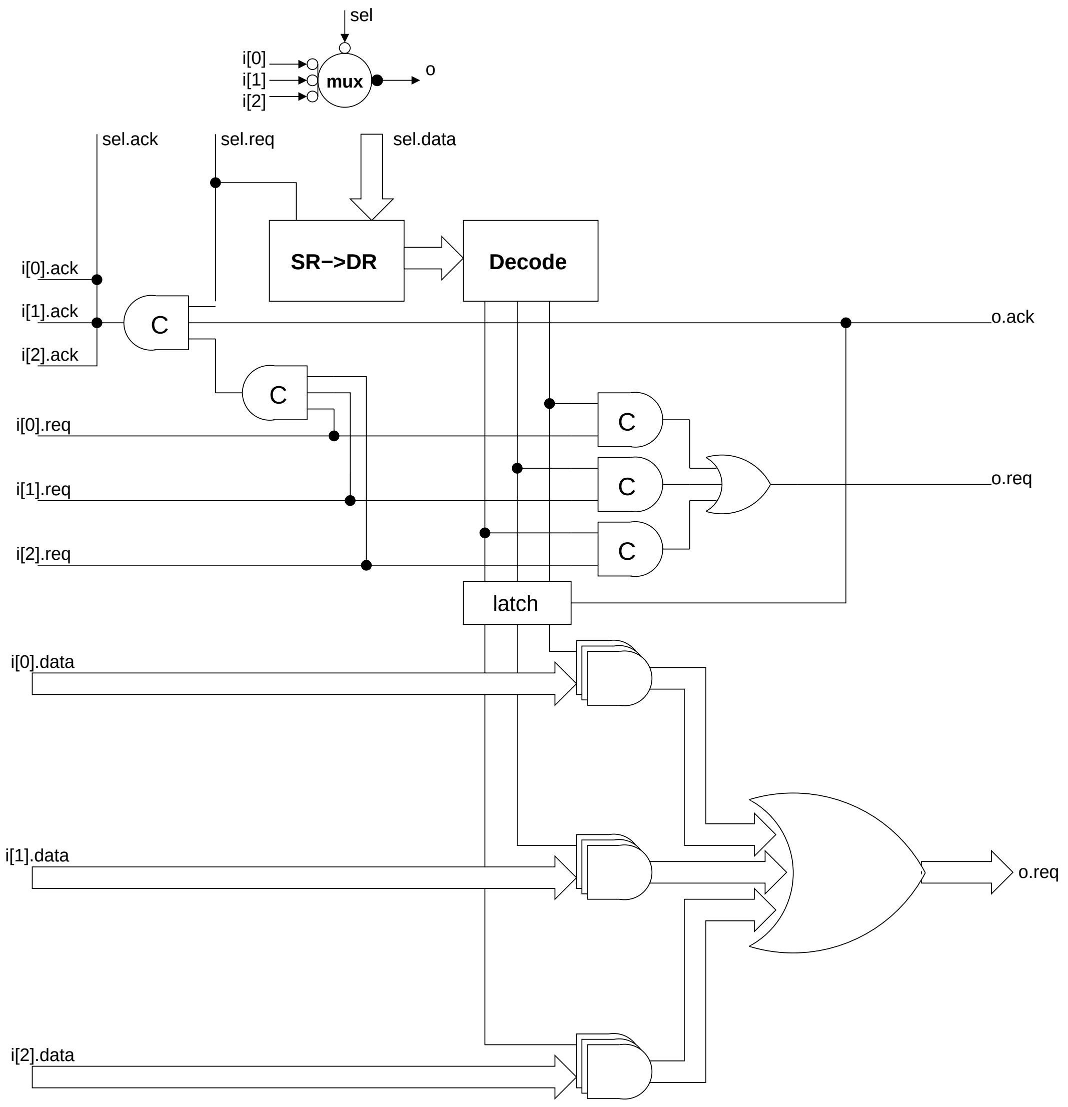


Figure B.23: Mux bundled-data circuit

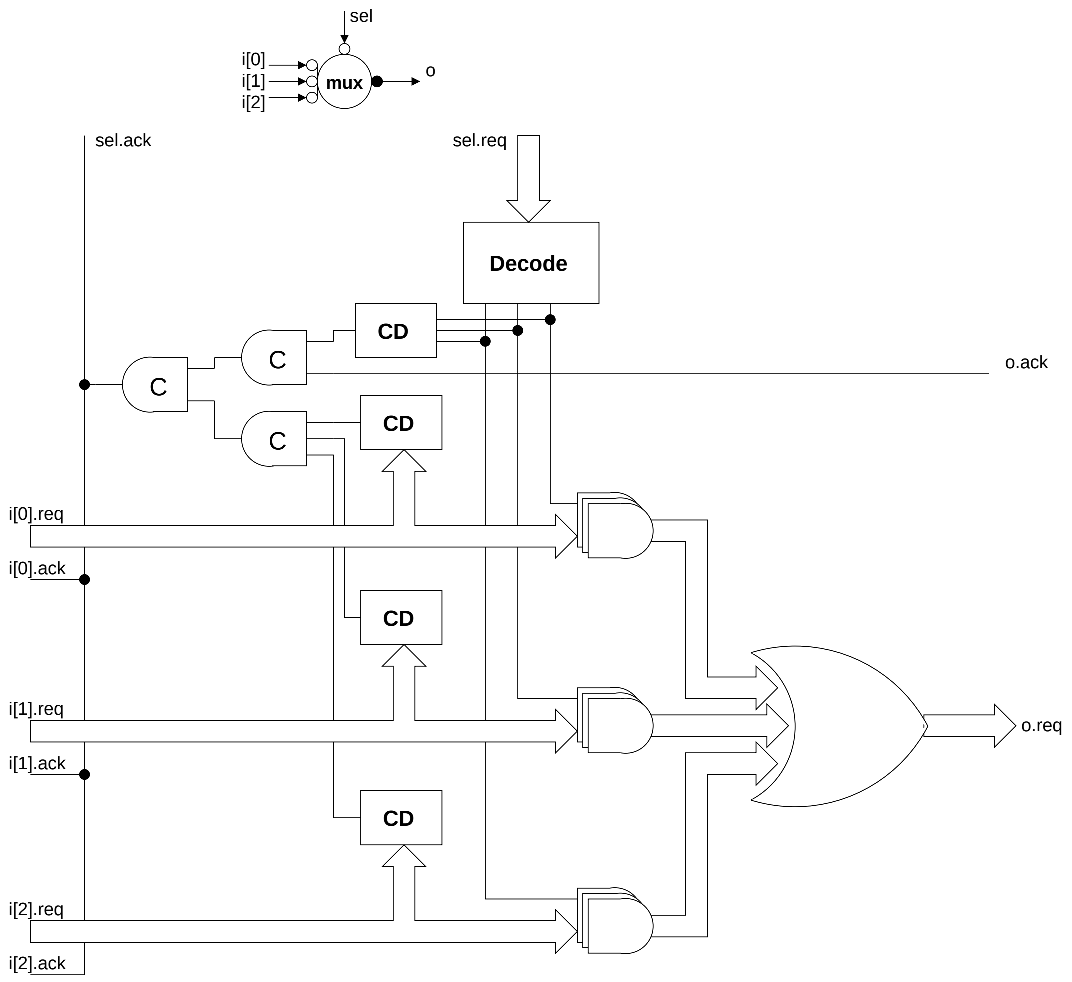


Figure B.24: Mux dual-rail circuit

Appendix C

Data-driven nanoSpa description

```
constant READPORTS = 3 -- number of readports in RegBank

type Colour is enumeration GREEN, RED end

-- memory interface types
type MemProcMode is enumeration USER=0, PRIVILEGED=1 end
type MemAbort    is bit
type MemRorW     is enumeration WRITE=0, READ=1 end
type MemIorD     is enumeration INSTRUCTION, DATA end
type MemAddress  is 32 bits
type MemSize     is enumeration BYTE=0, HALFWORD=1,
                                WORD=2, RESERVED=3 end
type MemData     is 32 bits
type MemLock     is bit
type MemSeq      is bit
type MemAccess   is record
        RorW    : MemRorW;
        Address : MemAddress;
        Size    : MemSize;
        Lock    : MemLock;
        Seq     : MemSeq;
        Colour  : Colour
end
type MemCtrl is record
    RorW : MemRorW;
    Size : MemSize;
    Lock : MemLock;
    Seq  : MemSeq
end

type Address is 32 bits
type Datapath is 32 bits
type Instruction is 32 bits

type ConditionCode is 4 bits

type RegNum  is 4 bits
type RegSpec is 5 bits
type RegDesc is 16 bits
type RegCnt  is 5 bits
type EncodedReg is record
    Num  : RegNum;
    Mask : RegDesc
end
type RegBank is bit

type WriteCtrl is bit

type Flags is record
    V : bit;
    C : bit;
    Z : bit;
    N : bit
end

type AluOpcode is enumeration
    AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC,
    TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN
end

type AluCtrl is AluOpcode
type AluSelect is 5 bits

type aBusSelect is 2 bits
type bBusSelect is 2 bits

type PcSelect is 2 bits

type ShiftType is 2 bits

procedure pipeReg (
    parameter DataType : type
    input  i : DataType
    output o : DataType      )
is
    variable x : DataType
begin
    input  i
    output x
    during
        x <- i
    end

    input  x
    output o
    during
        o <- x
    end
end

procedure pipeReg_Address is pipeReg(Address)
procedure pipeReg_Colour is pipeReg(Colour)
procedure pipeReg_Instruction is pipeReg(Instruction)
procedure pipeReg_ShiftType is pipeReg(ShiftType)
procedure pipeReg_AluCtrl is pipeReg(AluCtrl)
procedure pipeReg_MemCtrl is pipeReg(MemCtrl)
procedure pipeReg_bit is pipeReg(bit)
procedure pipeReg_2bits is pipeReg(2 bits)
procedure pipeReg_3bits is pipeReg(3 bits)
procedure pipeReg_5bits is pipeReg(5 bits)
procedure pipeReg_AluSelect is pipeReg(AluSelect)
procedure pipeReg_aBusSelect is pipeReg(aBusSelect)
procedure pipeReg_bBusSelect is pipeReg(bBusSelect)
procedure pipeReg_PcSelect is pipeReg(PcSelect)
procedure pipeReg_ConditionCode is pipeReg(ConditionCode)
procedure pipeReg_RegNum is pipeReg(RegNum)

procedure nanoSpaHarvard (
    -- the memory i/f:
    array 2 of output memAccess : MemAccess;
    array 2 of output memMode   : MemProcMode;
    array 2 of input  memDi     : MemData;
    array 1 of output memDo     : MemData;
    array 2 of input  memAbort  : MemAbort     )
is
    channel fiPc, foPc         : Datapath
    channel diPc, doPc         : Datapath
    channel xiPc, xoPc         : Datapath
    channel incI, incO         : Datapath
    channel diLr, doLr         : Datapath
    channel fiColour, foColour : Colour
    channel diColour, doColour : Colour
    channel xiColour, xoColour : Colour
    channel fiInstr, foInstr   : Instruction
    channel diBypass, doBypass : bit
    channel diShiftCtrl        : ShiftType
    channel doShiftCtrl        : ShiftType
    channel diShamt            : 5 bits
    channel doShamt            : 5 bits
    channel diAluCtrl          : AluCtrl
    channel doAluCtrl          : AluCtrl
    channel diSalu             : AluSelect
    channel diSalu3            : 2 bits
    channel diSalu0            : 2 bits
    channel doSalu             : AluSelect
    channel doSalu3            : 2 bits
    channel doSalu0            : 2 bits
    channel diSrds             : bit
    channel doSrds             : bit
    channel diSetFlags         : bit
    channel doSetFlags         : bit
    channel diSetMode          : bit
    channel doSetMode          : bit
    channel diSaveMode         : bit
    channel doSaveMode         : bit
    channel diWpc              : bit
    channel doWpc              : bit
    channel diSpc              : PcSelect
    channel doSpc              : PcSelect
    channel diSab              : bit
    channel diSab2             : bit
    channel doSab              : bit
    channel doSab2             : bit
    channel diStp              : bit
    channel diStp2             : bit
    channel doStp              : bit
    channel doStp2             : bit
    channel diMemCtrl          : MemCtrl
    channel doMemCtrl          : MemCtrl
    channel diWmem             : bit
    channel doWmem             : bit
    channel diMemOp            : bit
    channel doMemOp            : bit
    channel diSdi              : 2 bits
    channel doSdi              : 2 bits
    channel diCC, doCC         : ConditionCode
    channel diImm, doImm       : Datapath
    channel diMca, doMca       : aBusSelect
    channel diMcb, doMcb       : bBusSelect
    channel diMcr, doMcr       : bit
    channel diMcs, doMcs       : bit
    channel diMcpc, doMcpc     : bit
    array READPORTS of channel drai, drao : RegNum
    array READPORTS of channel drci, drco : bit
    array READPORTS of channel erb        : RegBank
    array READPORTS of channel era        : RegNum
    array READPORTS of channel erc        : bit
    array READPORTS of channel rdi        : Datapath
    array 2 of channel wai, wao           : RegNum
    array 2 of channel waci, waco         : bit
    variable wrb                          : RegBank
    variable wra                          : RegNum
    variable wd                           : Datapath
    variable wc                           : WriteCtrl
begin
    nanoFetch(memAccess[1], memMode[1], memDi[1], memAbort[1],
              fiPc, fiColour, fiInstr, xoPc, xoColour)

    pipeReg_Datapath(fiPc, foPc)
    pipeReg_Colour(fiColour, foColour)
    pipeReg_Instruction(fiInstr, foInstr)

    nanoDecode(foPc, foColour, foInstr,
               incI, incO, diPc, diLr, diColour,
               diBypass, diShiftCtrl, diShamt,
               diAluCtrl, diSalu, diSalu3, diSalu0, diSrds,
               diSetFlags, diSetMode, diSaveMode,
               diWpc, diSpc, diSab, diSab2, diStp, diStp2,
               diMemCtrl, diMemOp, diWmem, diSdi,
               diCC, diImm, diMca, diMcb, diMcr, diMcs, diMcpc,
               drai, drci, wai, waci)

    pipeReg_Datapath(diPc, doPc)
    pipeReg_Datapath(diLr, doLr)
    pipeReg_Colour(diColour, doColour)
    pipeReg_bit(diBypass, doBypass)
    pipeReg_ShiftType(diShiftCtrl, doShiftCtrl)
    pipeReg_5bits(diShamt, doShamt)
    pipeReg_AluCtrl(diAluCtrl, doAluCtrl)
    pipeReg_AluSelect(diSalu, doSalu)
    pipeReg_2bits(diSalu3, doSalu3)
    pipeReg_2bits(diSalu0, doSalu0)
    pipeReg_bit(diSrds, doSrds)
    pipeReg_bit(diSetFlags, doSetFlags)
    pipeReg_bit(diSetMode, doSetMode)
    pipeReg_bit(diSaveMode, doSaveMode)
    pipeReg_bit(diWpc, doWpc)
    pipeReg_PcSelect(diSpc, doSpc)
    pipeReg_bit(diSab, doSab)
    pipeReg_bit(diSab2, doSab2)
    pipeReg_bit(diStp, doStp)
    pipeReg_bit(diStp2, doStp2)
    pipeReg_MemCtrl(diMemCtrl, doMemCtrl)
    pipeReg_bit(diMemOp, doMemOp)
    pipeReg_bit(diWmem, doWmem)
    pipeReg_2bits(diSdi, doSdi)
    pipeReg_ConditionCode(diCC, doCC)
    pipeReg_Datapath(diImm, doImm)
    pipeReg_aBusSelect(diMca, doMca)
    pipeReg_bBusSelect(diMcb, doMcb)
    pipeReg_bit(diMcr, doMcr)
    pipeReg_bit(diMcs, doMcs)
    pipeReg_bit(diMcpc, doMcpc)

    pipeReg_RegNum(drai[0], drao[0])
    pipeReg_RegNum(drai[1], drao[1])
    pipeReg_RegNum(drai[2], drao[2])
    pipeReg_bit(drci[0], drco[0])
    pipeReg_bit(drci[1], drco[1])
    pipeReg_bit(drci[2], drco[2])
    pipeReg_RegNum(wai[0], wao[0])
    pipeReg_RegNum(wai[1], wao[1])
    pipeReg_bit(waci[0], waco[0])
    pipeReg_bit(waci[1], waco[1])

    nanoExecute(incI, incO, doPc, doLr, doColour,
                doBypass, doShiftCtrl, doShamt,
                doAluCtrl, doSalu, doSalu3, doSalu0, doSrds,
                doSetFlags, doSetMode, doSaveMode,
                doWpc, doSpc, doSab, doSab2, doStp, doStp2,
                doMemCtrl, doMemOp, doWmem, doSdi,
                doCC, doImm, doMca, doMcb, doMcr, doMcs, doMcpc,
                wao, waco, xiPc, xiColour,
                memAccess[0], memMode[0], memDi[0], memDo[0],
                memAbort[0], drao, drco, erb, era, erc, rdi,
                wrb, wra, wc, wd)

    nanoRegBank(erb, era, erc, wrb, wra, wc, wd, rdi)

    pipeReg_Address(xiPc, xoPc)
    pipeReg_Colour(xiColour, xoColour)
end

procedure nanoFetch (
```

```
    output memAccess  : MemAccess;
    output memMode    : MemProcMode;
    input  memDi      : MemData;
    input  memAbort   : MemAbort;

    -- the decoder i/f:
    output fPc        : Datapath;
    output fColour    : Colour;
    output fInstr     : Instruction;

    -- the execute i/f:
    input  xPc        : Datapath;
    input  xColour    : Colour       )
is
    type InstrSync is record
        pc    : Datapath
        instr : Instruction
    end

    variable pc, nextPc         : Datapath
    variable instr              : Instruction
    variable colour, nextColour : Colour
    variable goSeq              : bit
    channel  instrsync          : InstrSync
    sync seq
begin
    input  pc, instr
    output memAccess, nextPc, instrsync
    during
        memAccess <- ({READ, (pc as Address), WORD,
                              0, 1, GREEN} as MemAccess)
        instrsync <- ({pc, instr} as InstrSync)
        nextPc    <- (pc + 4 as Datapath)
    end

    input  instrsync
    output fPc, fInstr
    during
        fPc    <- instrsync.pc
        fInstr <- instrsync.instr
    end

    output memMode
    always
        memMode <- (PRIVILEGED as MemProcMode)
    end

    input memAbort
    during
        continue
    end

    input  memDi
    output instr
    during
        instr <- (memDi as Instruction)
    end

    input  nextPc, goSeq
    output pc
    during
        case goSeq of
          1 then
            pc <- nextPc
        | 0 then
            input xPc during pc <- xPc end
        end
    end

    input  colour
    output fColour, nextColour, goSeq
    during
        fColour <- colour
        arbitrate seq then
            goSeq <- 1
            nextColour <- colour
        | xColour then
            goSeq <- 0
            nextColour <- xColour
        end
    end

    input  nextColour
    output colour
    during
        colour <- nextColour
    end

    output   seq
    always
       sync seq
    end

    init
        pc     <- (0 as Datapath)
        colour <- (0 as Colour)
    end
end

procedure nanoDecode (
    -- the fetch i/f:
    input  fPc       : Address;
    input  fColourIn : Colour;
    input  fInstrIn  : Instruction;

    -- the execute i/f:
    input  incI      : Datapath;
    output incO      : Datapath;
    output dPc       : Datapath;
    output dLr       : Datapath;
    output dColour   : Colour;
    output bypass    : bit;
    output shiftCtrl : ShiftType;
    output shamt     : 5 bits;
    output aluCtrl   : AluCtrl;
    output salu      : AluSelect;
    output salu3     : 2 bits;
    output salu0     : 2 bits;
    output srds      : bit;
    output setFlags  : bit;
    output setMode   : bit;
    output saveMode  : bit;
    output wpc       : bit;
    output spc       : PcSelect;
    output sab       : bit;
    output sab2      : 1 bits;
    output stp       : bit;
    output stp2      : 1 bits;
    output dMemCtrl  : MemCtrl;
    output memOp     : bit;
    output wmem      : bit;
    output sdi       : 2 bits;
    output cc        : ConditionCode;
    output immediate : Datapath;
    output mca       : aBusSelect;
    output mcb       : bBusSelect;
    output mcr       : bit;
    output mcs       : bit;
    output mcpc      : bit;

    -- the register bank i/f:
    array READPORTS of output ra : RegNum;
    array READPORTS of output rc : bit;
    array 2 of output wa         : RegNum;
    array 2 of output wc         : bit    )
is
    variable sendLr    : bit
    variable noshift   : bit
    variable incIn     : Datapath
    variable incOut    : Datapath
    variable incOp     : Datapath

    variable lsmMw     : 6 bits
    variable lsmMr     : 6 bits
    channel  lsmM      : 6 bits
    variable rBaseMw   : RegNum
    variable rBaseMr   : RegNum
    channel  rBaseM    : RegNum
    variable regDesc   : RegDesc
    variable regCount  : RegCnt
    channel  regCountS : RegCnt
    variable regCountw : RegCnt
    variable regCountr : RegCnt
    variable regNum    : RegNum
    variable last      : bit
    variable isFirst   : bit
    variable goAgain   : 1 bits
    variable incSel    : 1 bits
    variable fInstr    : Instruction
    variable fInstrw   : Instruction
    variable fColour   : Colour
    variable fColourw  : Colour
begin
    init
        fInstrw   <- (0 as Instruction)
        fColourw  <- (0 as Colour)
        lsmMw     <- (0 as 6 bits)
        rBaseMw   <- (0 as RegNum)
        regCountw <- (0 as RegCnt)
        goAgain   <- (0 as 1 bits)
    end

    input  goAgain, fInstrw, fColourw,
           lsmMw, rBaseMw, regCountw
    output fInstr, fColour, lsmMr, rBaseMr,
           regCountr, isFirst, regDesc
    during
        case goAgain of
          1 then
            fInstr    <- fInstrw
            fColour   <- fColourw
            lsmMr     <- lsmMw
            rBaseMr   <- rBaseMw
            regCountr <- regCountw
            isFirst   <- (0 as bit)
        | 0 then
            input fInstrIn
            during
                fInstr <- fInstrIn
                lsmMr <- (#fInstrIn[20..24] @
                          #fInstrIn[15..15] as 6 bits)
                rBaseMr <- (#fInstrIn[16..19] as RegNum)

                case (#fInstrIn[25..27] as 3 bits) of
                  0b00x, 0b01x, 0b101, 0b11x then -- not ldm/stm
                    regCountr <- (0 as RegCnt)
                | 0b100 then
                    regDesc <- (#fInstrIn[0..15] as RegDesc)
                    isFirst <- (1 as bit)
                    input regCount during
                        regCountr <- regCount
                    end
                end
            end
            input fColourIn during
                fColour <- fColourIn
            end
        end
    end

    input  fInstr, fColour, noshift, lsmMr, rBaseMr, regCountr
    output dColour, cc, sendLr, incSel, noshift, bypass, shiftCtrl,
           shamt, mcs, srds, aluCtrl, salu, salu0, salu3, setFlags,
           setMode, wpc, spc, sab, sab2, dMemCtrl, memOp, wmem, sdi,
           immediate, mca, mcb, mcr, mcpc, rc, ra, wc, wa,
           fInstrw, lsmMw, rBaseMw, regCountw, lsmM, rBaseM,
           regCountS, saveMode, stp, stp2, goAgain, fColourw
    during
        fInstrw    <- fInstr
        fColourw   <- fColour
        lsmMw      <- lsmMr
        rBaseMw    <- rBaseMr
        regCountw  <- regCountr

        case (#fInstr[25..27] as 3 bits) of
          0b00x, 0b01x, 0b101, 0b11x then
            goAgain <- 0
            <decode regular instruction>
        | 0b100 then
            case <last> of
              1 then
                goAgain <- 0
            else
                goAgain <- 1
            end
            <decode load/store multiple>
        end
    end
end

constant NUMREGS = 16
constant SUPREGL = 13
constant SUPREGH = 14

procedure nanoRegBank (
    -- the read port i/f (inputs)
    array READPORTS of input rbk : RegBank;
    array READPORTS of input ra  : RegNum;
    array READPORTS of input rc  : bit;

    -- the write port i/f
    input wbk                    : RegBank;
    input wa                     : RegNum;
    input wc                     : WriteCtrl;
    input wd                     : Datapath;
    -- the read port i/f (outputs)
    array READPORTS of output rd : Datapath  )
is
    array NUMREGS of variable reg_usrw : Datapath
    array NUMREGS of variable reg_usrr : Datapath
    array SUPREGL..SUPREGH of variable reg_svcr : Datapath
    array SUPREGL..SUPREGH of variable reg_svcw : Datapath

    type WriteBack is record
        wc  : WriteCtrl
        wa  : RegNum
        wbk : RegBank
    end

    variable wc0  : WriteBack
    variable wc1  : WriteBack
    variable wd0  : Datapath
    variable wd1  : Datapath
    variable write_num : 1 bits
    variable next_num : 1 bits

    array READPORTS of channel ind : log NUMREGS bits
begin
    input  reg_usrr, reg_svcr, rc, ind
    output rd, reg_usrw, reg_svcw, ind
    during
        foreach i in rd
            case rc[i] of
              1 then
                input ra[i], rbk[i]
                during
                    ind[i] <- (ra[i] as log NUMREGS bits)
                    case rbk[i] of
                      1 then
                        rd[i] <- reg_svcr[(#(ind[i])[1])]
                    | 0 then
                        rd[i] <- reg_usrr[ind[i]]
                    end
                end
            | 0 then
                ind[i] <- (0 as log NUMREGS bits)
            end
        end

        foreach i in reg_usrw
            reg_usrw[i] <- reg_usrr[i]
        end
        foreach i in reg_svcw
            reg_svcw[i] <- reg_svcr[i]
        end
    end
```

```
input  reg_usrw, reg_svcw, wc0, wc1, wd0, wd1
output reg_usrr, reg_svcr
during
    foreach i in reg_usrr
        case wc0 @ wc1 of
          0b1 @ (i as RegNum) @ 0b0 @ 0b0 @ 0bxxxxx,
          0b1 @ (i as RegNum) @ 0b0 @ 0b1 @
                    not #(i as RegNum)[0] @ 0bxxxx,
          0b1 @ (i as RegNum) @ 0b0 @ 0b1 @
              0bx @ not #(i as RegNum)[1] @ 0bxxx,
          0b1 @ (i as RegNum) @ 0b0 @ 0b1 @
             0bxx @ not #(i as RegNum)[2] @ 0bxx,
          0b1 @ (i as RegNum) @ 0b0 @ 0b1 @
            0bxxx @ not #(i as RegNum)[3] @ 0bx
          then
            reg_usrr[i] <- wd0
        |  0bxxxxxx @ 0b1 @ (i as RegNum) @ 0b0 then
            reg_usrr[i] <- wd1
        else
            reg_usrr[i] <- reg_usrw[i]
        end
    end

    foreach i in reg_svcr
        case wc0 @ wc1 of
          0b1 @ (i as RegNum) @ 0b1 @ 0b0 @ 0bxxxxx,
          0b1 @ (i as RegNum) @ 0b1 @ 0b1 @
                    not #(i as RegNum)[0] @ 0bxxxx,
          0b1 @ (i as RegNum) @ 0b1 @ 0b1 @
              0bx @ not #(i as RegNum)[1] @ 0bxxx,
          0b1 @ (i as RegNum) @ 0b1 @ 0b1 @
             0bxx @ not #(i as RegNum)[2] @ 0bxx,
          0b1 @ (i as RegNum) @ 0b1 @ 0b1 @
            0bxxx @ not #(i as RegNum)[3] @ 0bx
          then
            reg_svcr[i] <- wd0
        | 0bxxxxxx @ 0b1 @ (i as RegNum) @ 0b1 then
            reg_svcr[i] <- wd1
        else
            reg_svcr[i] <- reg_svcw[i]
        end
    end
end

input  write_num, wc
output next_num, wc0, wc1, wd0, wd1
during
    next_num <- not write_num
    case wc of
      1 then
        input wa, wbk, wd during
            case write_num of
              0 then
```

```
                  wc0 <- ({1, wa, wbk} as WriteBack)
                wd0 <- wd
            | 1 then
                wc1 <- ({1, wa, wbk} as WriteBack)
                wd1 <- wd
            end
        end
    | 0 then
        case write_num of
          0 then
            wc0 <- ({0,0,0} as WriteBack)
            wd0 <- (0 as Datapath)
        | 1 then
            wc1 <- ({0,0,0} as WriteBack)
            wd1 <- (0 as Datapath)
        end
    end

  end

  input  next_num
  output write_num
  during
      write_num <- next_num
  end

  init
      write_num <- 0
      all reg_usrr <- (0 as Datapath)
      all reg_svcr <- (0 as Datapath)
  end
end

procedure nanoExecute (
  -- the decode i/f:
  output incI           : Datapath;
  input  incO           : Datapath;
  input  dPc            : Datapath;
  input  dLr            : Datapath;
  input  dColour        : Colour;
  input  bypass         : bit;
  input  shiftCtrl      : ShiftType;
  input  shamt          : 5 bits;
  input  aluCtrl        : AluCtrl;
  input  salu           : AluSelect;
  input  salu3          : 2 bits;
  input  salu0          : 2 bits;
  input  srds           : bit;
  input  setFlags       : bit;
  input  setMode        : bit;
  input  saveMode       : bit;
  input  wpc            : bit;
  input  spc            : PcSelect;
  input  sab            : bit
  input  sab2           : 1 bits;
  input  stp            : bit;
  input  stp2           : 1 bits;
  input  memCtrl        : MemCtrl;
  input  memOp          : bit;
  input  wmem           : bit;
  input  sdi            : 2 bits;
  input  cc             : ConditionCode;
  input  immediate      : Datapath;
  input  mca            : aBusSelect;
  input  mcb            : bBusSelect;
  input  mcr            : bit;
  input  mcs            : bit;
  input  mcpc           : bit;
  array 2 of input  wra : RegNum;
  array 2 of input  wrc : bit;

  -- the fetch i/f:
  output xPc            : Datapath;
  output xColour        : Colour;

  -- the memory i/f:
  output memAccess      : MemAccess;
  output memMode        : MemProcMode;
  input  memDi          : MemData;
  output memDo          : MemData;
  input  memAbort       : MemAbort;

  -- the register bank i/f:
  array READPORTS of input  dra : RegNum;
  array READPORTS of input  drc : bit;
  array READPORTS of output erb : RegBank;
  array READPORTS of output era : RegNum;
  array READPORTS of output erc : bit;
  array READPORTS of input  rd  : Datapath;
  output wb                     : RegBank;
  output wa                     : RegNum;
  output wc                     : WriteCtrl;
  output wd                     : Datapath  )
is
  variable memAddress     : MemAddress
  variable pc0, pc1       : Datapath
  variable eColour        : Colour
  variable cFlags, aFlags : Flags
  variable aluFlags       : Flags
  variable shiftIn        : Datapath
  variable rdl            : Datapath
  variable rsl            : 5 bits
  variable aBusIn, aBus   : Datapath
  variable aBusD          : Datapath
  channel  aBusT          : Datapath
  variable bBus           : Datapath
```

```
    variable aluResult      : Datapath
    variable aluResult0     : Datapath
    variable aluResult1     : Datapath
    variable aluResult2     : Datapath
    variable aluResult3     : Datapath
    variable incOut0        : Datapath
    variable incOut1        : Datapath
    channel  tempIn         : Datapath
    channel  tempOut        : Datapath
    array 2 of channel rwb  : Datapath
    variable rwbIn          : Datapath
    channel  rwbOut         : Datapath
    variable kma, kmc, kmd  : bit
    variable sdiK           : 2 bits
    array 2 of variable kra : 2 bits
    array 2 of variable krc : 1 bits
    variable krd            : bit
    variable kpc            : bit
    variable rm, sm         : bit
    variable smm            : bit
    variable uc, uf         : bit
    variable memDi0         : Datapath
    channel  memDi1         : Datapath
    variable shDst          : 5 bits
    variable fi             : bit
    variable fo             : bit
    variable memMd          : MemProcMode
    array READPORTS of variable modeR : MemProcMode
    array 2         of variable modeW : MemProcMode
begin
    nanoExecuteControl(cc, dColour, wpc, setFlags, setMode, saveMode,
                       wrc, memOp, wmem, sdi, cFlags, eColour,
                       kma, kmc, kmd, kra, krc, krd, kpc, sdiK,
                       rm, sm, smm, uc, uf)

    colour(uc, eColour, xColour)

    mode(rm, sm, smm, memMd, modeR, modeW)

    bankSel(drc[0], dra[0], modeR[0], erc[0], era[0], erb[0])

    bankSel(drc[1], dra[1], modeR[1], erc[1], era[1], erb[1])

    bankSel(drc[2], dra[2], modeR[2], erc[2], era[2], erb[2])

    steerPc(dPc, spc, pc0, pc1)

    steerRds(rd[2], srds, rsl, rdl)

    mux3_aBus(rd[1], pc1, aBusT, mca, aBusIn)

    selFork_Datapath(aBusIn, sab, aBus, aBusD)

    mux3_bBus(immediate, pc0, rd[0], mcb, shiftIn)

    mux2_5bits(rsl, shamt, mcs, shDst)

    nanoShifter(bypass, shiftIn, fi, shiftCtrl, shDst, bBus, fo)

    nanoAlu(aBus, bBus, aluCtrl, aFlags, fo, aluResult, aluFlags)

    flags(aluFlags, uf, cFlags, aFlags, fi)

    steerAlu(aluResult, salu,
             aluResult0, aluResult1, aluResult2, aluResult3, incI)

    mux2_Datapath(dLr, aluResult1, mcr, rwbIn)

    selStop_Datapath(rwbIn, krd, rwbOut)

    pipeReg_Datapath(rwbOut, rwb[0])

    regWriteBack(modeW, wra, rwb, kra, krc, wb, wa, wc, wd)

    mux2_pc(aluResult2, memDi0, mcpc, kpc, xPc)

    selFork_Datapath(incO, stp, incOut0, incOut1)

    input  salu3, stp2
    output tempIn
    during
        case (#salu3 @ #stp2 as 3 bits) of
          0bxx1 then
            input aluResult3 during
                tempIn <- aluResult3
            end
        | 0b110 then
            input incOut1 during
                tempIn <- incOut1
            end
        else continue
        end
    end

    pipeReg_Datapath(tempIn, tempOut)

    pipeReg_Datapath(tempOut, aBusT)

    input  salu0, sab2
    output memAddress
    during
        case (#salu0 @ #sab2 as 3 bits) of
          0bxx1 then
            input aluResult0 during
                memAddress <- (aluResult0 as MemAddress)
            end
        | 0bx10 then
            input incOut0 during
                memAddress <- (incOut0 as MemAddress)
            end
        | 0b100 then
            input aBusD during
                memAddress <- (aBusD as MemAddress)
            end
        else continue
        end
    end

    memAccess(memAddress, memCtrl, rdl, memMd, memDi0, memDi1,
              sdiK, kma, kmc, kmd, memAccess, memMode, memDo,
      memDi, memAbort)

    pipeReg_Datapath(memDi1, rwb[1])
end

procedure nanoAlu (
    input  a    : Datapath;
    input  b    : Datapath;
    input  ctrl : AluCtrl;
    input  fi   : Flags;
    input  sfc  : bit; -- carry from shifter
    output o    : Datapath;
    output f    : Flags    )
is
    type AddArg is record
        ci  : bit;
        arg : Datapath
    end

    type AluResult is record
        result : Datapath;
        co     : bit
    end

    -- used to take the relevant bits of the output of the adder
    type AddResult is record
        _ : bit; -- the sum of the carry ins, always zero
        result : AluResult
    end

    variable a_v, b_v : Datapath
    variable ci_v     : bit
    variable v1, v2   : bit
    variable result   : AluResult
    variable aXORb    : bit
    channel  ctrl0    : AluCtrl
    variable ctrl1    : AluCtrl
    variable ctrl2    : AluCtrl
    channel  fi0      : Flags
    variable fi1      : Flags
    variable c        : bit
    variable n        : bit
begin
    input  ctrl
    output ctrl0, ctrl1, ctrl2
    during
        ctrl0 <- ctrl
        ctrl1 <- ctrl
        ctrl2 <- ctrl
    end

    input  fi
    output fi0, fi1
    during
        fi0 <- fi
        fi1 <- fi
    end

    input  ctrl0, fi0, a, b, sfc
    output a_v, b_v, ci_v
    during
        case ctrl0 of
          RSB, RSC then
            a_v <- not a
        | AND, EOR, SUB, ADD, ADC, SBC, TST,
          TEQ, CMP, CMN, ORR, MOV, BIC, MVN then
            a_v <- a
        end

        case ctrl0 of
          SUB, SBC, MVN, BIC, CMP then
            b_v <- not b
        | AND, EOR, RSB, ADD, ADC, RSC,
          TST, TEQ, CMN, ORR, MOV then
            b_v <- b
        end

        case ctrl0 of
          ADC, SBC, RSC then
            ci_v <- fi0.C
        | SUB, RSB, CMP then
            ci_v <- 1
        | CMN, ADD then
            ci_v <- 0
        | AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN then
            ci_v <- sfc
        end
    end

    input  ctrl1, a_v, b_v, ci_v
    output result, aXORb
    during
```

```
        aXORb <- #(a_v)[sizeof Datapath-1] xor
                 #(b_v)[sizeof Datapath-1]
        case ctrl1 of
          SUB, RSB, CMP, ADD, CMN, ADC, SBC, RSC then
            result <- (
            ({ci_v, a_v} as (sizeof AddArg) bits) +
            ({ci_v, b_v} as (sizeof AddArg) bits)
            as AddResult).result
        | MOV, MVN then
            result <- ({b_v, ci_v} as AluResult)
        | EOR, TEQ then
            result <- ({a_v xor b_v, ci_v} as AluResult)
        | ORR then
            result <- ({a_v or b_v, ci_v} as AluResult)
        | AND, BIC, TST then
            result <- ({a_v and b_v, ci_v} as AluResult)
        end
    end

    input  ctrl2, fi1, result, aXORb
    output o, v1, v2, c, n
    during
        o <- result.result

        case ctrl2 of
          SUB, RSB, CMP, ADD, CMN, ADC, SBC, RSC then
            v1 <- (result.co xor
                   #(result.result)[sizeof Datapath-1] xor aXORb)
        | AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN then
            v1 <- fi1.V
        end

        v2 <- (result.result = 0 as bit)

        c <- result.co

        n <- #(result.result)[sizeof Datapath-1]
    end

    input  v1, v2, c, n
    output f
    during
        f <- ({v1, c, v2, n} as Flags)
    end
end

procedure nanoExecuteControl (
    -- the decode inputs:
    input  cc            : ConditionCode;
    input  dColour       : Colour;
    input  wpc           : bit;
    input  setFlags      : bit;
    input  setMode       : bit;
```

```
    input  saveMode        : bit;
    array 2 of input wac   : bit;
    input  memOp           : bit;
    input  wmem            : bit;
    input  sdiIn           : 2 bits;

    -- the execute inputs:
    input  flags           : Flags;
    input  eColour         : Colour;

    -- the control outputs:
    output kma    : bit
    output kmc    : bit
    output kmd    : bit
    array 2 of output kra : 2 bits;
    array 2 of output krc : 1 bits;
    output krd    : bit;
    output kpc    : bit;
    output sdiOut : 2 bits;
    output rm     : bit;
    output sm     : bit;
    output smm    : bit;
    output uc     : bit
    output uf     : bit                     )
is
    variable stop : bit
begin
    input  cc, dColour, flags, eColour
    output stop
    during
        case (#flags[0..3] @ #cc[0..3] @
              #dColour[0..0] @ #eColour[0..0] as 10 bits) of
          0b000000x1xx, 0b110000x1xx,
          0b000001x0xx, 0b110001x0xx,
          0b000010xx1x, 0b110010xx1x,
          0b000011xx0x, 0b110011xx0x,
          0b0001001xxx, 0b1101001xxx,
          0b0001010xxx, 0b1101010xxx,
          0b000110xxx1, 0b110110xxx1,
          0b000111xxx0, 0b110111xxx0,
          0b001000x01x, 0b111000x01x,
          0b001001x10x, 0b111001x10x,
          0b0010100xx0, 0b1110100xx0,
          0b0010101xx1, 0b1110101xx1,
          0b0010110xx1, 0b1110110xx1,
          0b0010111xx0, 0b1110111xx0,
          0b00110000x0, 0b11110000x0,
          0b00110010x1, 0b11110010x1,
          0b001101x1xx, 0b111101x1xx,
          0b00110100x1, 0b11110100x1,
          0b00110110x0, 0b11110110x0,
          0b001110xxxx, 0b111110xxxx then
            stop <- 0
    else
        stop <- 1
    end
end

input  stop, setFlags, setMode, saveMode, wpc, wac, memOp, wmem
output uf, kpc, rm, sm, uc, kra,
       smm, kmc, kma, sdiOut, kmd, krd, krc
during
    uf <- (not stop) and setFlags
    rm <- (not stop) and setMode
    sm <- (not stop) and saveMode
    uc <- (not stop) and wpc
    smm <- (not stop) and memOp

    kra[0] <- (#stop[0..0] @ #(wac[0])[0..0] as 2 bits)
    kra[1] <- (#stop[0..0] @ #(wac[1])[0..0] as 2 bits)

    case wpc of
      1 then
        kpc <- stop
    | 0 then
        continue
    end

    case wac[0] of
      1 then
        krd <- stop
        case stop of
          1 then
            krc[0] <- 0b1
        | 0 then
            krc[0] <- 0b0
        end
    | 0 then
        krc[0] <- 0b1
    end

    case wac[1] of
      1 then
        case stop of
          1 then
            krc[1] <- 0b1
        | 0 then
            krc[1] <- 0b0
        end
    | 0 then
        krc[1] <- 0b1
    end

    case memOp of
      1 then
        kmc <- stop
            kma <- stop

            case wmem of
              0 then
                input sdiIn during
                    case stop of
                      0 then
                        sdiOut <- sdiIn
                    | 1 then
                        continue
                    end
                end
            | 1 then
                continue
            end
    | 0 then
        continue
    end

    case wmem of
      1 then
        kmd <- stop
    | 0 then
        continue
    end
end
end


procedure pipeReg_Datapath is pipeReg(Datapath)

procedure forkA (
    parameter DataType : type
    input  i  : DataType
    output o0 : DataType
    output o1 : DataType      )is
begin
    input  i
    output o0, o1
    during
        o0 <- i
        o1 <- i
    end
end

procedure fork_Datapath is forkA(Datapath)

procedure mux3 (
    parameter DataType : type;
    parameter CtrlType : type;
    input  a    : DataType;
    input  b    : DataType;
    input  c    : DataType;
    input  ctrl : CtrlType;
    output o    : DataType    ) is
begin
    input  ctrl
    output o
    during
        case (ctrl as 3 bits) of
          0b00 then
            input a during o <- a end
        | 0b01 then
            input b during o <- b end
        | 0b11 then
            input c during o <- c end
        else continue
        end
    end
end

procedure mux3_aBus is mux3(Datapath, aBusSelect)
procedure mux3_bBus is mux3(Datapath, bBusSelect)

procedure steerDi (
  input  a    : MemData;
  input  ctrl : 2 bits;
  input  sel  : 2 bits;
  output o0   : Datapath;
  output o1   : Datapath  )
is
    channel inna : array 4 of 8 bits
begin
    input  ctrl, a
    output o0, o1, inna
    during
        case ctrl of
        0b00 then
            o0 <- (a as Datapath)
        | 0b01 then
            o1 <- (a as Datapath)
        | 0b1x then
            inna <- (a as array 4 of 8 bits)

            input sel, inna during
                case ctrl of
                  0b10 then
                    o0 <- (inna[sel] as Datapath)
                else
                    o1 <- (inna[sel] as Datapath)
                end
            end
        end
    end
end
```

```
procedure steerAlu (
    input  a    : Datapath;
    input  ctrl : AluSelect;
    output o0   : Datapath;
    output o1   : Datapath;
    output o2   : Datapath;
    output o3   : Datapath;
    output o4   : Datapath  ) is
begin
    input  ctrl, a
    output o0, o1, o2, o3, o4
    during
        case (#ctrl[0..0] as bit) of
          1 then
            o0 <- a
        | 0 then continue
        end

        case (#ctrl[1..1] as bit) of
          1 then
            o1 <- a
        | 0 then continue
        end

        case (#ctrl[2..2] as bit) of
          1 then
            o2 <- a
        | 0 then continue
        end

        case (#ctrl[3..3] as bit) of
          1 then
            o3 <- a
        | 0 then continue
        end

        case (#ctrl[4..4] as bit) of
          1 then
            o4 <- a
        | 0 then continue
        end
    end
end

procedure steerPc (
    input  a    : Datapath;
    input  ctrl : PcSelect;
    output o0   : Datapath;
    output o1   : Datapath ) is
begin
    input  ctrl, a
    output o0, o1
    during
```

```
        case (#ctrl[0..0] as bit) of
          1 then
            o0 <- a
        | 0 then continue
                end
                case (#ctrl[1..1] as bit) of
          1 then
            o1 <- a
        | 0 then continue
        end
    end
end

procedure steerRds (
    input  a    : Datapath;
    input  ctrl : bit;
    output o0   : 5 bits;
    output o1   : Datapath ) is
begin
    input  ctrl, a
    output o0, o1
    during
        case ctrl of
          1 then
            o1 <- a
        else
            o0 <- (a as 5 bits)
        end
    end
end

procedure bankSel (
    input  rcI   : bit;
    input  rNumI : RegNum;
    input  pMode : MemProcMode;

    output rcO   : bit;
    output rNumO : RegNum;
    output rBk   : RegBank     )
is
    channel pModeInner : MemProcMode
begin
    input  rcI, pMode
    output rcO, rNumO, rBk, pModeInner
    during
        rcO <- rcI
        case rcI of
          1 then
            pModeInner <- pMode

            input rNumI, pModeInner during
                case (#rNumI[0..3] @
                      #pModeInner[0..0] as 5 bits) of
                      0b0xxxx,    -- usr regs
                      0b10000, 0b10001, 0b10010,
                      0b10011, 0b10100, 0b10101,
                      0b10110, 0b10111, 0b11000,
                      0b11001, 0b11010, 0b11011,
                      0b11100, -- "unshadowed" svc regs
                      0b11111 then   -- should not happen !!
                        rBk <- (0 as RegBank)
                    | 0b11101, 0b11110 then
                        rBk <- (1 as RegBank)
                    end

                    rNumO <- rNumI
                end
            | 0 then continue
            end
        end
    end

procedure bankSelStop (
  input  krn   : 2 bits;
  input  rNumI : RegNum;
  input  pMode : MemProcMode;

  output rNumO : RegNum;
  output rBk   : RegBank     )
is
    channel pModeInner : MemProcMode
begin
    input  krn, pMode
    output rNumO, rBk, pModeInner
    during
        case krn of
          0b0x then
            continue
        | 0b1x then
            pModeInner <- pMode

            input rNumI, pModeInner during
                case krn of
                  0b10 then
                    case (#rNumI[0..3] @
          #pModeInner[0..0] as 5 bits) of
                      0b0xxxx,    -- usr regs
                      0b10000, 0b10001, 0b10010,
                      0b10011, 0b10100, 0b10101,
                      0b10110, 0b10111, 0b11000,
                      0b11001, 0b11010, 0b11011,
                      0b11100, -- "unshadowed" svc regs
                      0b11111 then   -- should not happen !!
                        rBk <- (0 as RegBank)
                    | 0b11101, 0b11110 then
                        rBk <- (1 as RegBank)
                    end

                    rNumO <- rNumI
                else continue
                end
            end
        end
    end
end

procedure selFork (
    parameter DataType : type;
    input  a    : DataType;
    input  ctrl : bit;
    output o0   : DataType;
    output o1   : DataType    ) is
begin
    input  ctrl, a
    output o0, o1
    during
        o0 <- a
        case ctrl of
          1 then
            o1 <- a
        | 0 then continue
        end
    end
end

procedure selFork_Datapath is selFork(Datapath)

procedure selStop (
    parameter DataType : type;
    input  a    : DataType;
    input  ctrl : bit;
    output o    : DataType    ) is
begin
    input  ctrl, a
    output o
    during
        case ctrl of
          0 then
            o <- a
        | 1 then continue
        end
    end
end

procedure selStop_MemCtrl is selStop(MemCtrl)
procedure selStop_Datapath is selStop(Datapath)
procedure selStop_Address is selStop(Address)
procedure selStop_RegNum is selStop(RegNum)
```

```
procedure mode (
    input  rm                       : bit;
    input  sm                       : bit;
    input  sMemMode                 : bit;
    output memMode                  : MemProcMode;
    array READPORTS of output modeR : MemProcMode;
    array 2         of output modeW : MemProcMode )
is
    variable cmode    : MemProcMode
    variable cmoderet : MemProcMode
    variable smode    : MemProcMode
    variable updMode  : 2 bits
    variable smoderet : MemProcMode
begin
    init
        cmode    <- (PRIVILEGED as MemProcMode)
        smoderet <- (USER as MemProcMode)
    end

    input  cmode, smoderet, sMemMode, sm, rm
    output smode, updMode, modeR, modeW, memMode, cmoderet
    during
        foreach i in modeR
            modeR[i] <- cmode
        end

        modeW[1] <- cmode

        case sMemMode of
          1 then
            memMode <- cmode
        | 0 then continue
        end

        case sm of
          1 then
            smode <- cmode
            modeW[0] <- (PRIVILEGED as MemProcMode)
        else
            modeW[0] <- cmode
            smode <- smoderet
        end

        updMode <- (#rm[0..0] @ #sm[0..0] as 2 bits)

        cmoderet <- cmode
    end

    input  updMode, smode, cmoderet
    output cmode, smoderet
    during
        case updMode of
          0b00 then
```

```
            cmode <- cmoderet
        | 0b01 then
            cmode <- smode
        | 0b1x then
            cmode <- (PRIVILEGED as MemProcMode)
        end
        smoderet <- smode
    end

end

procedure flags (
    input  fi  : Flags;
    input  uf  : bit;
    output fo0 : Flags;
    output fo1 : Flags;
    output sfc : bit   )
is
    variable flags, nextFlags : Flags
    variable flags2           : Flags
    variable upd              : bit
begin
    init
        flags <- (0 as Flags)
    end

    input  flags
    output fo0, fo1, sfc, flags2
    during
        fo0 <- flags
        fo1 <- flags
        sfc <- flags.C
        flags2 <- flags
    end

    input  uf
    output upd
    during
        upd <- uf
    end

    input  flags2, upd, fi
    output nextFlags
    during
        case upd of
          1 then
              nextFlags <- fi
        | 0 then
            nextFlags <- flags2
        end
    end

    input  nextFlags
```

```
    output flags
    during
        flags <- nextFlags
    end
end

procedure colour (
    input  upd  : bit;
    output co0  : Colour;
    output co1  : Colour )
is
    variable colour, nextColour : Colour
begin
    init
        colour <- (GREEN as Colour)
    end

    input  upd, colour
    output co0, co1, nextColour
    during
            co0 <- colour
        case upd of
          1 then
            co1 <- not colour
            nextColour <- not colour
        else
            nextColour <- colour
        end
    end

    input  nextColour
    output colour
    during
        colour <- nextColour
    end
end

-- regWriteBack multiplexes the 2 writes to the register bank.
-- regBank has a single write port so writes must be sequenced.
-- (writes can be skipped with na or nb)
procedure regWriteBack (
    array 2 of input pmd  : MemProcMode;
    array 2 of input raIn : RegNum;
    array 2 of input rd   : Datapath;
    array 2 of input kra  : 2 bits;
    array 2 of input krc  : 1 bits;
    output ob  : RegBank;
    output oa  : RegNum;
    output oc  : WriteCtrl;
    output od  : Datapath                )
is
    array 2 of variable rb : RegBank
    array 2 of variable ra : RegNum
    variable sel, selnext  : bit
begin
    init
        sel <- 0
    end

    bankSelStop(kra[0], raIn[0], pmd[0], ra[0], rb[0])

    bankSelStop(kra[1], raIn[1], pmd[1], ra[1], rb[1])

    input  sel
    output oc, ob, od, oa, selnext
    during
        selnext <- not sel

        case sel of
          0 then
              input krc[0] during
                case krc[0] of
                  1 then
                    oc <- (0b0 as WriteCtrl)
                | 0 then
                    oc <- (0b1 as WriteCtrl)
                    input rb[0] during
                        ob <- rb[0]
                    end
                    input ra[0] during
                        oa <- ra[0]
                    end
                    input rd[0] during
                        od <- rd[0]
                    end
                end
            end
        | 1 then
            input krc[1] during
                case krc[1] of
                  1 then
                    oc <- (0b0 as WriteCtrl)
                | 0 then
                    oc <- (0b1 as WriteCtrl)
                    input rb[1] during
                        ob <- rb[1]
                    end
                    input ra[1] during
                        oa <- ra[1]
                    end
                    input rd[1] during
                        od <- rd[1]
                    end
                end
            end
        end
    end

    input  selnext
    output sel
    during
        sel <- selnext
    end
end

procedure memAccess (
    -- the internal data i/f
    input  memAddressIn : Address;
    input  memCtrlIn    : MemCtrl;
    input  memDoRd      : Datapath;
    input  memMd        : MemProcMode;
    output memDi0       : Datapath;
    output memDi1       : Datapath;
    -- the control control i/f
    input  sdiK         :2 bits;
    input  kma          : bit;
    input  kmc          : bit;
    input  kmd          : bit;
    -- the memory i/f
    output memAccess    : MemAccess;
    output memMode      : MemProcMode;
    output memDo        : Datapath;
    input  memDi        : Datapath;
    input  memAbort     : MemAbort    )
is
    channel memAddress : Address
    channel memCtrl    : MemCtrl
    channel memDiIn    : Datapath
    channel memDoIn    : Datapath
    channel byteSel    : 2 bits
begin
    selStop_Datapath(memDoRd, kmd, memDoIn)

    selStop_Address(memAddressIn, kma, memAddress)

    selStop_MemCtrl(memCtrlIn, kmc, memCtrl)

    memDiReg(memDi, memDiIn)

    steerDi(memDiIn, sdiK, byteSel, memDi0, memDi1)

    input  memAddress, memCtrl
    output memAccess, memDo, byteSel
    during
        memAccess <- ({memCtrl.RorW, memAddress, memCtrl.Size,
               memCtrl.Lock, memCtrl.Seq, GREEN} as MemAccess)

        case (#memCtrl.Size[1..1] @ #memCtrl.RorW[0..0] as 2 bits) of
        0b0x then -- str
```

```
            input memDoIn during
                case #memCtrl.Size[1..1] of -- strb
                  0 then
                    memDo <- (#memDoIn[0..7] @ #memDoIn[0..7] @
              #memDoIn[0..7] @ #memDoIn[0..7] as Datapath)
                | 1 then -- str
                    memDo <- memDoIn
                end
            end
        | 0b10 then -- ldrb
            byteSel <- (#memAddress[0..1] as 2 bits)
        | 0b11 then -- ldr
            continue
        end
    end

    input memMd output memMode during memMode <- memMd end

    input memAbort during continue end
end
```

Appendix D

Balsa Handshake Components

This appendix provides a very brief summary of the components from the conventional Balsa component set that are featured in this thesis. The intention is to provide a key to the symbols and an informal description of the behaviour.

The Balsa component set can be roughly split into three categories.

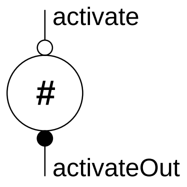
Control components use only sync ports. They feature an Activation port that is used to start the operation of the component. Output sync channels are then connected to the activation ports of other components.

Datapath component deal only with channels carrying data. They are used for storing, processing, merging and splitting data channels.

Control / datapath interface components are used to control the movement of data through the datapath. They have one or more sync ports used to communicate with control components as well as data channel ports. They initiate handshakes on data channels in response to activations or issue activations on receipt of data.

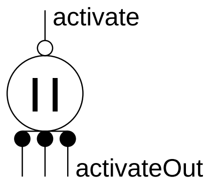
D.1 Control components

D.1.1 Loop



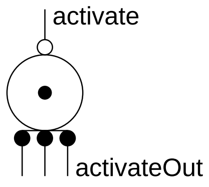
The Loop component is used to implement infinite repetition. Once activated it produces an infinite sequence of activations on its output port.

D.1.2 Concur



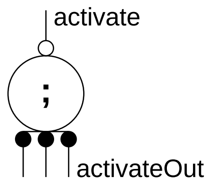
The Concur component produces an activation on all of its output ports following an input activation. All the output activations are begun at the same time but then operate independently.

D.1.3 Fork



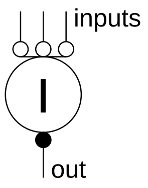
The Fork component produces an activation on all of its output ports following an input activation. In a four-phase protocol, all the outputs synchronise between the processing and return-to-zero phases. See also figure 2.18 on page 52.

D.1.4 Sequence



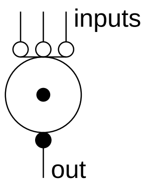
The Sequence component is similar to the Concur component but its output activations are produced one at at time in sequence.

D.1.5 Call



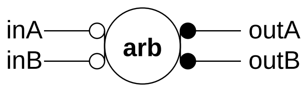
The Call component passes a handshake on one of its input ports to the output port. The inputs must not occur concurrently.

D.1.6 Sync



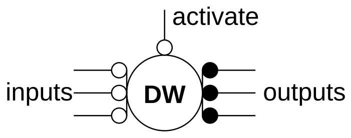
The Sync component synchronises the request on all of its inputs before passing these handshakes to the output.

D.1.7 Arbitrate



Arbitrate passes a handshake on inA to outA or a handshake on inB to outB. If both inA and inB are activated concurrently it makes a non-deterministic decision as to which to pass first.

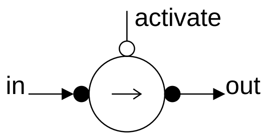
D.1.8 DecisionWait



DecisionWait synchronises an activation with one of its inputs and then passes this handshake to the corresponding output. The inputs must be mutually exclusive.

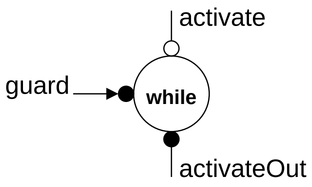
D.2 Control / datapath interface components

D.2.1 Fetch



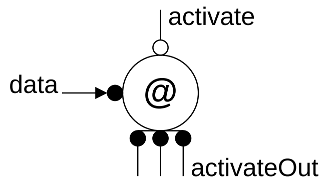
Upon activation the Fetch component pulls data on its input port and then pushes it on the output.

D.2.2 While



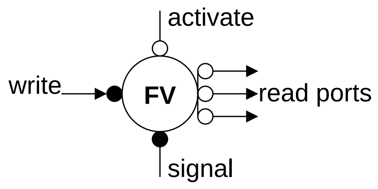
The While component is used to implement the guarded loop language construct. When it is activated the While component pulls a single bit data item from its ‘guard’ port. If the guard is true then While produces an output activation. When this activation has been acknowledged, While pulls another guard and repeats the process until a guard that is false is received.

D.2.3 Case



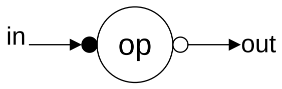
Upon activation, the Case component pulls a guard on its data port. It then activates one of its outputs based on the data that was received. Multiple values can be mapped to each output. If some values are not mapped to an output they will result in no output activation.

D.2.4 FalseVariable



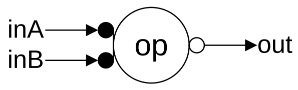
Upon activation, the FalseVariable pulls data on its write port. It then holds this handshake open and activates the signal port. Then FalseVariable acts as a Variable component, supplying, on request, the data from the write port to a set of read ports. When the signal handshake is completed, the write data is released.

D.2.5 UnaryFunc



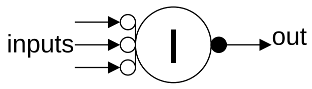
Implements single-operand operations such as invert. The handshake is simply passed through the component with the modified data.

D.2.6 BinaryFunc



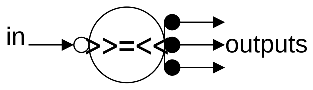
BinaryFunc is used to implement two-operand operations such as addition, subtraction, comparisons and bit-wise boolean functions. The output request is forked to both inputs. The input acknowledges are synchronised and passed to the output.

D.2.7 CallMux



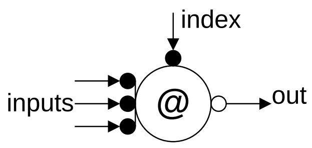
CallMux is used as a merge element in datapaths. Multiple push input channels can are merged onto a single output channel. The inputs must be mutually exclusive.

D.2.8 SplitEqual



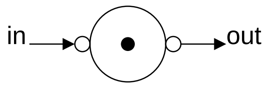
SplitEqual splits the data on its input port to multiple chunks of the same width, one chunk being sent on each output.

D.2.9 CaseFetch



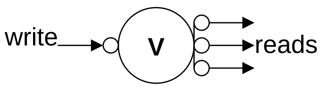
When CaseFetch receives a request on its output, it pulls an index and uses this to decide which of its input ports to pull data on and then passes this data to the output port.

D.2.10 PassivatorPush



PassivatorPush is used to connect an active output port from one process to the active input port of another process.

D.2.11 Variable



The Variable component has a single write port and multiple read ports. It stores data that it receives on the write port and provides it to the read ports on request. Reads and writes must not occur concurrently.

References

[AML97] S. Appleton, S. Morton, and M. Liebelt. High performance two-phase asynchronous pipelines. *IEICE Transactions on Information and Systems*, E80-D(3):287–295, March 1997.

[Bar00] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000.

[BB96] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[BBK$^+$94] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[BE97] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

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

[BJN99] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[BRD95] G. M. Birtwistle, C. J. Van Rijsbergen, and A.L. Davis, editors. *Asynchronous Digital Circuit Design (Workshops in Computing)*. Springer-Verlag Telos, April 1995.

[Bre06] Charles Brej. *Early Output Logic and Anti-Tokens*. PhD thesis, The University of Manchester, 2006.

[BSVMH84] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[CKK$^+$97] 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.

[CKLS06] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[CLA] High-speed asynchronous pipeline technology for the CLASS project. http://www.cs.unc.edu/Grants/Abstracts/KT3408.html.

[Cla67] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336. Academic Press, 1967.

[CN02] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

[CNBE02] T. Chelcea, S. Nowick, A. Bardsley, and D. Edwards. A burst-mode oriented back-end for the balsa synthesis system. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 330. IEEE Computer Society, 2002.

[EBJ$^+$06] Doug Edwards, Andrew Bardsley, Lilian Janin, Luis Plana, and Will Toms. *Balsa: A Tutorial Guide*. The University of Manchester, May 2006.

[Esp] Espresso. http://www-cad.eecs.berkeley.edu/Software.

[FDG$^+$93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[FES00] S. B. Furber, A. Efthymiou, and Montek Singh. A power-efficient duplex communication system. In Alex Yakovlev and Reinder Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 145–150, July 2000.

[FGG98] S. B. Furber, James D. Garside, and David A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, October 1998.

[FGT$^+$97] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, April 1997.

[FNT$^+$99] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[Fri01] E. Friedman. Clock distribution networks in synchronous digital integrated circuits, 2001.

[GBvB$^+$98] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80C51 microcontroller. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 96–107, 1998.

[GG97] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, April 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[KL02] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Design & Test of Computers*, 19(4):107–117, 2002.

[KPWK02] Joep Kessels, Ad Peeters, Paul Wielage, and Suk-Jin Kim. Clock synchronization through handshake signalling. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 59–68, April 2002.

[KVL96] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Asynchronous Circuits and Systems*, March 1996.

[Liu97] J. Liu. *Arithmetic and control components for an asynchronous microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.

[MAC$^+$02] Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins, and George Taylor. Improving smart card security using self-timed circuits. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 211–218, April 2002.

[Mar90] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[MBL$^+$89] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.

[MLM$^+$97] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[MM98] Rajit Manohar and Alain J. Martin. Slack elasticity in concurrent computing. In J. Jeuring, editor, *Proc. 4th International Conference on the Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 272–285, 1998.

[Mul62] David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.

[NS94] Lars Skovby Nielsen and Jens Sparsø. Low-power operation using self-timed and adaptive scaling of the supply voltage. In *1994 International Workshop on Low Power, Napa, California*, April 1994.

[NSJ90] C. D. Nielsen, J. Staunstrup, and S. R. Jones. Potential performance advantages of delay insensitivity. In *Proceedings of Workshop on Silicon Architectures for Neural Nets*, November 1990.

[NUK$^+$94] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[PDF$^+$92] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, pages 351–355. IEEE Computer Society Press, October 1992.

[PDF$^+$98] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 32–42, 1998.

[Pee96] Ad M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[PET$^+$07] Luis Plana, Doug Edwards, Sam Taylor, Luis Tarazona, and Andrew Bardsley. Performance-driven syntax-directed synthesis of asynchronous processors. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES)*, September 2007.

[PM06] Piyush Prakash and Alain J. Martin. Slack matching quasi delay-insensitive circuits. In *Proc. International Symposium on Asynchronous Circuits and Systems*, page 195. IEEE Computer Society, 2006.

[PN98] Luis A. Plana and Steven M. Nowick. Architectural optimization for low-power nonpipelined asynchronous systems. *IEEE Transactions on VLSI Systems*, 6(1):56–65, March 1998.

[PRB$^+$03] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, S. Temple, J. D. Garside, and Z. C. Yu. SPA— a secure Amulet core for smartcard applications. *Microprocessors and Microsystems*, 27(9):431–446, October 2003.

[PTE05] Luis A. Plana, Sam Taylor, and Doug Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *Proc. International Conf. Computer Design (ICCD)*, pages 703–710. IEEE Computer Society Press, October 2005.

[RVR99] M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 135–144, April 1999.

[SF01] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic, December 2001.

[SJ00] David Seal and David Jagger, editors. *ARM Architecture Reference Manual*. Addison Wesley, December 2000.

[SKC$^+$99] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev. Bridging modularity and optimality: delay-insensitive interfacing in asynchronous circuits synthesis. In *IEEE*

International Conference on Systems, Man, and Cybernetics, pages 899–904, 1999.

[SN01] Montek Singh and Steven M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. International Conf. Computer Design (ICCD)*, pages 9–17, November 2001.

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

[TAS] TIMA Asynchronous Synthesis Tools. http://tima.imag.fr/cis/.

[WM01] Catherine G. Wong and Alain J. Martin. Data-driven process decomposition for the synthesis of asynchronous circuits. In *IEEE International Conference on Electronics, Circuits and Systems*, 2001.

[WM03] Catherine G. Wong and Alain J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Proc. ACM/IEEE Design Automation Conference*, pages 508–513, June 2003.

[WPS95] Ted Williams, Niteen Patkar, and Gene Shen. SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor. *IEEE J. of Solid-State Circuits*, 30(11):1215–1226, November 1995.

[YBA96] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.