

# Dual-Rail False Variable Redesign

Luis A. Plana    Doug Edwards    Andrew Bardsley  
 Department of Computer Science  
 University of Manchester

The **FalseVariable (FV)** handshake component, shown in Figure 1, resembles a normal Variable with one passive write port  $WD$  and a number of passive read ports  $RD_i$ . It differs, however, in the presence of an active ‘probe’ port  $S$ . The component is named **FalseVariable** because it does not store data.

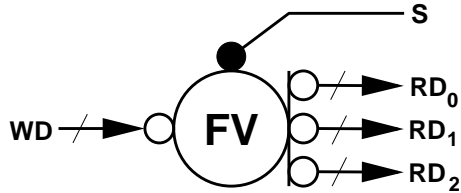


Figure 1: False Variable Handshake Component.

This component is used to implement the input **select** command. A simple example of the use of this command is an unbuffered multiplexer, described in Balsa as follows:

```
-- mux: unbuffered multiplexer
procedure mux (
  input a, b : byte;
  output c : byte) is
begin
  loop
    -- a and b behave like variables
    -- within their guarded command
    select a then c <- a
      | b then c <- b
    end
  end
end
```

Figure 2 shows how procedure *mux* is implemented. A **FV** is used in every input to create the read-only variable behaviour of channels within the **select** command.

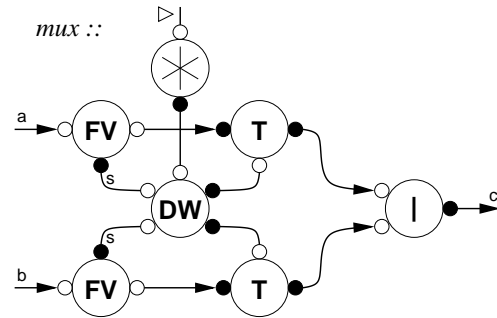


Figure 2: Procedure *mux* Implementation.

## FV Behaviour

The behaviour of the **FV** component can be described as follows: A **WRITER** process produces data that is pushed on channel  $WD$ . One or more **READER** processes consume data by pulling it on channels  $RD_i$ . The **READERs** must wait until valid data has been sent by the **WRITER** before reading. Channel  $S$  is used by **FV** to indicate the arrival of valid data on channel  $WD$ . Since **FV** does not store data, the **WRITER** is allowed to take the data away only after the **READER** has consumed it. All channels are implemented using *request* and *acknowledge* signals.

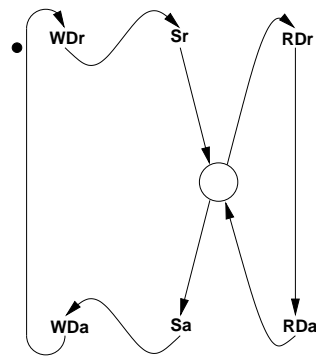


Figure 3: 2-Phase FalseVariable Behaviour.

Figure 3 shows the behaviour of **FV** with channels that follow a 2-phase protocol. For simplicity, the STG shows a single **READER**. The behaviour specifies that the **READER** can execute any number of read operations on **FV**. It also specifies that the **READER** is not required to consume data.

Figure 4(a) shows a possible 4-phase **FV** behaviour. In this protocol, the *request* and *acknowledge* signals must return to their original state to complete the handshake. This behaviour is “equivalent” to the 2-phase behaviour because the reader always completes the handshake before generating an event in  $S_a$ . The STG has been annotated indicating when data is valid on the  $WD$  and  $RD_i$  channels. Clearly, the specified behaviour is safe since data-validity on  $WD$  completely encloses data-validity on  $RD_i$ . A more general behaviour, shown in Figure 4(b), is achieved if the **READER** is allowed to generate  $S_a+$  as soon as it has finished reading the data for the last time. This behaviour of the **READER** may allow more concurrent operation and is the one most commonly implemented in Balsa.

Figure 5 shows a simplified version of the behaviour shown in Figure 4(b), assuming that the **READER** consumes the data once. The figure highlights the processing and return-to-zero (r-t-z) phases of the **WRITER** and **READER** processes. It is clear from the figure that these four phases are sequenced. Safe operation requires that the two processing phases be sequenced: the **WRITER** must produce data before **READER**s can consume it. However, the r-t-z phases can execute concurrently since, in both environments, the data is returning to the spacer state. This opens the opportunity for performance improvements wherever **FV** is used.

Figure 6(a) introduces a new, concurrent behaviour for the **FV** component. In this case, the **WRITER** is allowed to start its r-t-z phase as soon as the **READER** processing phase is finished, indicated by  $S_a+$ , allowing the two r-t-z phases to execute concurrently. However, this STG does not reflect the operation correctly.

The fact that **FV** does not store the data introduces additional dependencies, shown in Figure 6(b). If the **WRITER** takes the data away [ $WD_r-$ ] the **READER** data will return to spacer [ $RD_a-$ ] even if the **READER** has not closed the *read port*.

This is represented in the graph in Figure 6(b) by the *ORed* transitions leading to  $RD_a-$ : Either  $RD_r-$  or  $WD_r-$ , whichever happens first, will cause  $RD_a-$ . Clearly, this is not delay-insensitive behaviour. The synchronisation in  $WD_a-$  guarantees that the events in the path [ $S_a+ \Rightarrow WD_a+ \Rightarrow WD_r- \Rightarrow RD_a-$ ] always take place but the two highlighted events [ $S_r-$  and  $RD_r-$ ] may never occur. They can be “overtaken” by the events in the other path. Although this behaviour is the most concurrent, it is NOT safe.

Figure 7(a) shows an alternative, less concurrent behaviour for **FV**. In this case, the **WRITER** is allowed to withdraw its data only after the **READER** has closed the *read port* and the data has gone back to spacer [ $RD_a-$ ]. Although the **WRITER** r-t-z phase is started later than in the previous case, this behaviour is safe and also allows the two r-t-z phases to execute concurrently. However, implementing this behaviour may be costly: it requires **FV** to do completion detection on the outputs of all the *read ports* in order to detect  $RD_a-$ . This suggests the behaviour in Figure 7(b) as a cheaper alternative.

In this case the request to close the *read ports* [ $RD_r-$ ] is used instead of the outputs [ $RD_a-$ ] to allow the **WRITER** to start its r-t-z phase. This behaviour re-introduces the  $WD_r- \Rightarrow RD_a-$  dependency, with *OR* causality. However, the situation is different from the previous case since no events can be “overtaken”. The two *ORed* transitions leading to  $RD_a-$  correspond to the two branches of a fork in the  $RD_r-$  signal inside **FV**. The problem can be avoided by designating it as an *isochronic* fork. If this assumption stands, the  $WD_r- \Rightarrow RD_a-$  transition can be eliminated from the graph to obtain a valid STG. [Incidentally, this is a very reasonable assumption since the branch of the fork that has to “win” the race is an internal wire while the other branch corresponds to a signal that goes out to the **WRITER** and returns after the execution of its r-t-z phase].

Further performance improvements are possible by modifying the way *completion detection* (*CD*) is done in **FV**. Figure 8(a) shows the places where data is checked for completion: **WRITER** data is checked in **FV** after both processing and r-t-z phases while **READER** *CD* is carried out by the **READER**. The **READER**s must do *CD* in both the

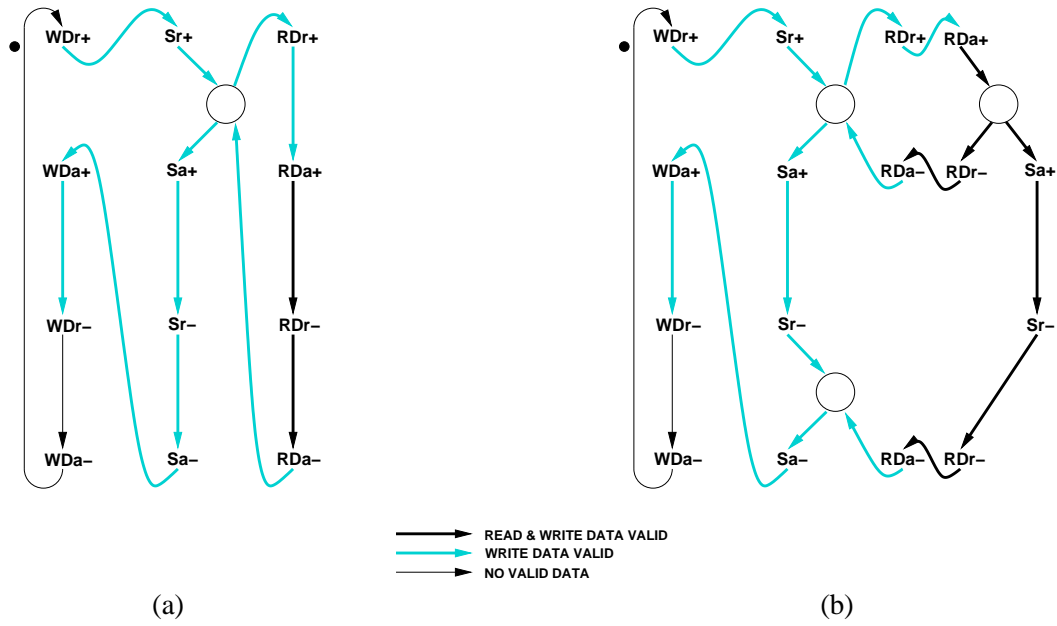


Figure 4: Alternative 4-Phase Behaviours.

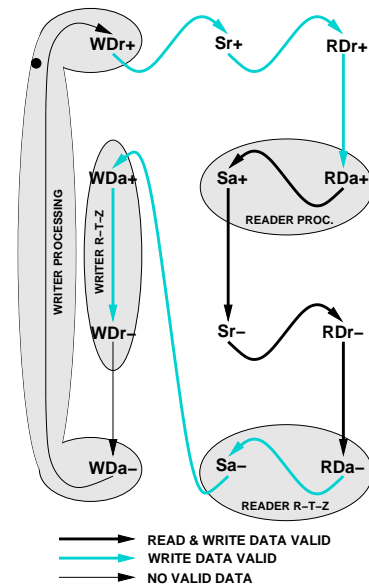


Figure 5: Processing and R-T-Z Phases.

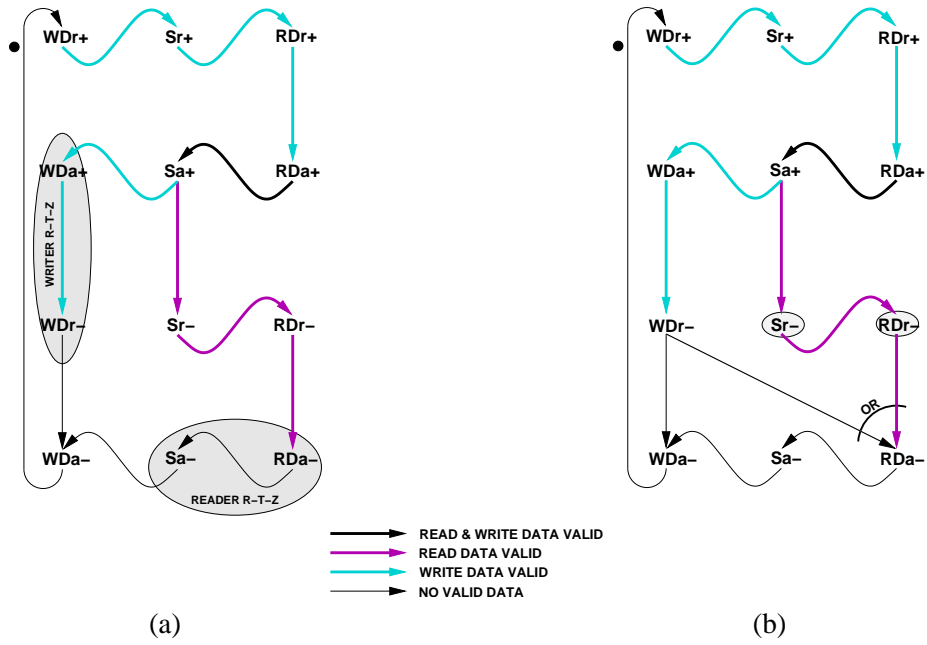


Figure 6: Concurrent FV Behaviour.

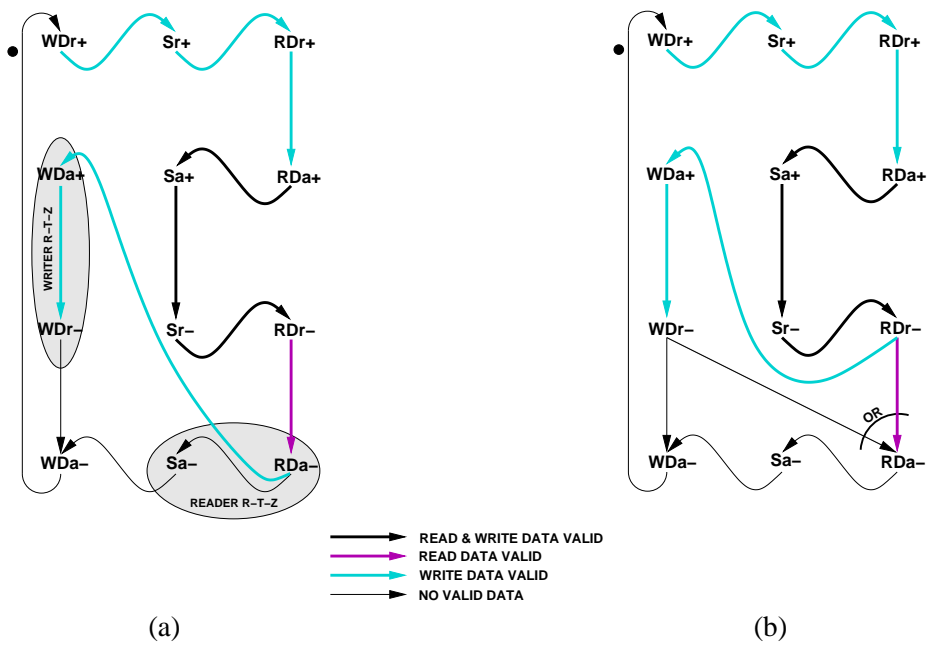
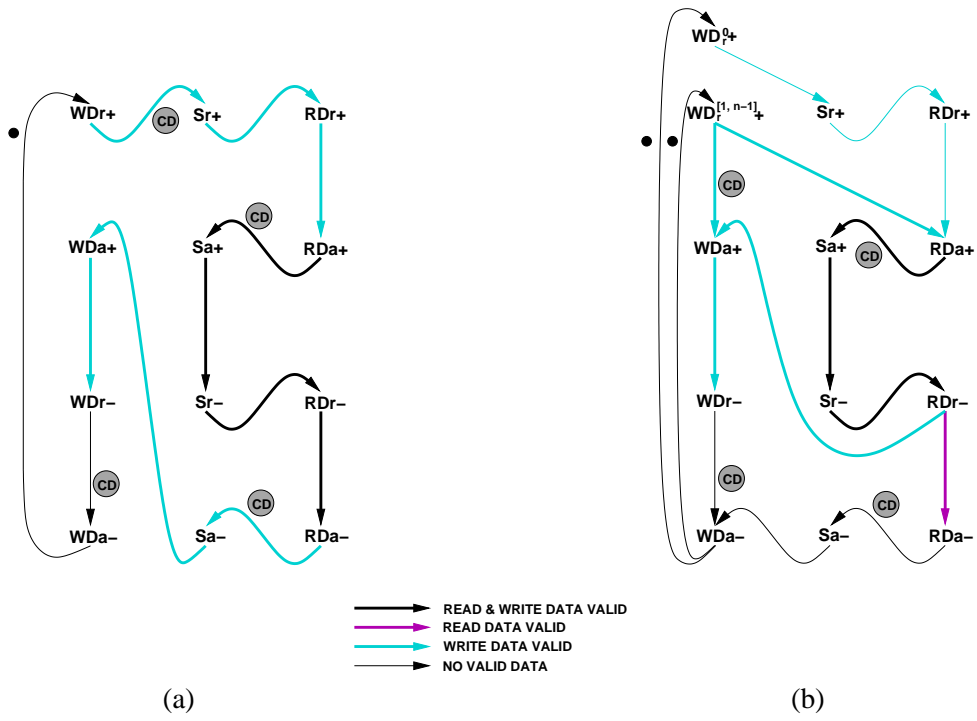
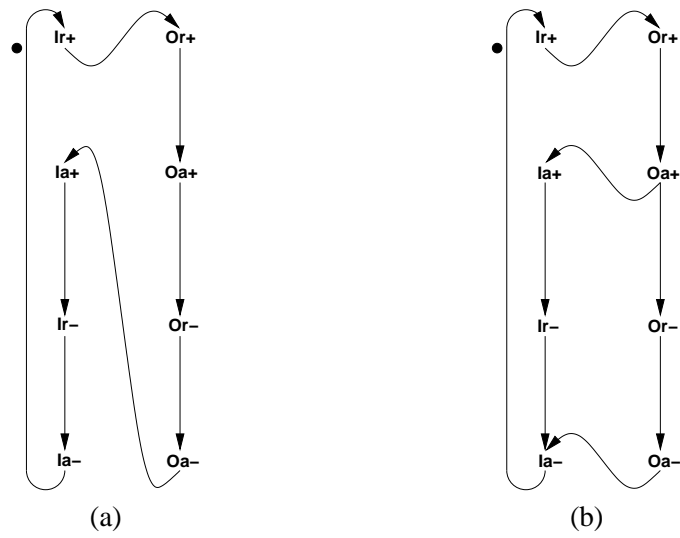


Figure 7: Safe Concurrent Behaviours.



(a) (b)  
 Figure 8: Existing and New FalseVariable Behaviours.



(a) (b)  
 Figure 9: S-element and T-element Behaviours.

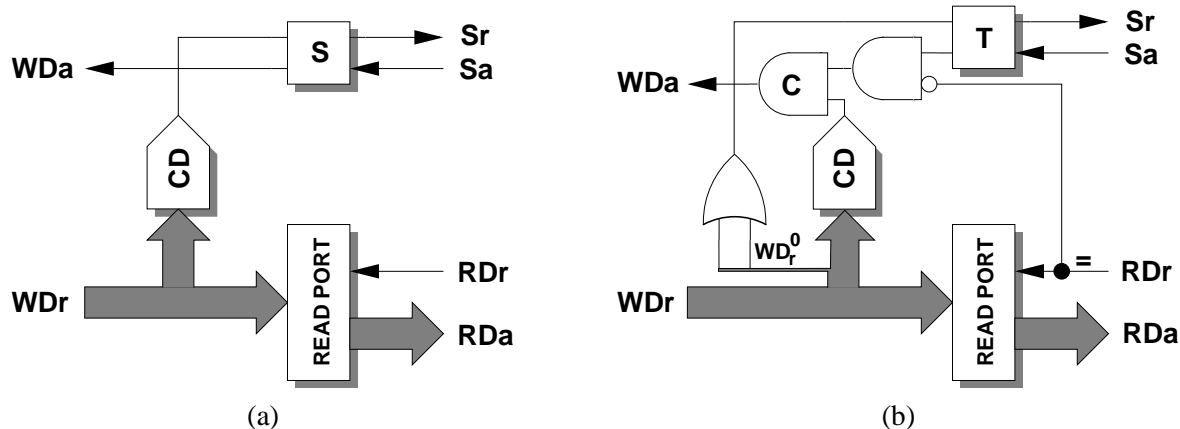


Figure 10: Current and Proposed **FV** Implementations.

processing and r-t-z phases to guarantee DI operation. In general, READER data  $CD$  would make WRITER data  $CD$  redundant. However, as indicated earlier, the READERS are not required to consume data. They may do it conditionally and, when they don't, no READER data  $CD$  will be carried out. To guarantee DI operation in all cases **FV** must verify  $CD$  on WRITER data.

Figure 8(a) shows that these four instances of  $CD$  are sequenced. If data is wide,  $CD$  can be very expensive both in time and area. The new concurrent behaviour executes the two r-t-z phases concurrently, which will result in concurrent  $CD$  on these phases. Figure 8(b) shows how the concurrent **FV** behaviour is modified to allow concurrent  $CD$  also on the processing phases. A single WRITER data bit is used to detect that data is present and trigger the events in  $S_r$ . This means that the READER may open the *read port*  $[RD_r+]$  before all the WRITER data is present but this is safe behaviour since READER data will remain in spacer state until WRITER data arrives, represented by the  $[WD_r+ \Rightarrow RD_a+]$  dependency. The new **FalseVariable** component implements the behaviour shown in Figure 8(b).

## FV Implementation

Figure 10(a) shows the current implementation of **FV**, corresponding to the behaviour in Figure 8(a). This implementation uses an *S-element* to achieve the sequential interleaving of events in the  $WD$  and

$S$  channels. The behaviour of the *S-element* is depicted in Figure 9(a).

Completion detection ( $CD$ ) is usually implemented with a tree of *C-elements*. A *read port* consists of an *AND* gate for every data signal, with the other input connected to the read request  $[RD_r]$ .

The proposed implementation for **FV** corresponding to the new, more concurrent behaviour is shown in Figure 10(b). The behaviour of the *T-element* used in this implementation is depicted in Figure 8(b).

The *isochronic* fork required for correct behaviour of the new **FV** is identified in Figure 10(b). One branch of  $RD_r-$  goes directly into the *read port* while the other goes through several gates to generate  $WD_a+$ , which will start the WRITER r-t-z phase that will generate  $WD_r-$ , finally reaching the *read port*. This is, clearly, a very reasonable *isochronic* fork assumption.

The proposed implementation of the **FV** component may result in a significant performance increase in dual-rail balsa-synthesised circuits.