The Formal Verification of the Fairisle ATM Switching Element

Paul Curzon

Technical Report Number 329

March 1994

University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
United Kingdom

The Formal Verification of the Fairisle ATM Switching Element

Paul Curzon

University of Cambridge,
Computer Laboratory
Email: pc@cl.cam.ac.uk

Abstract

We describe the formal verification of an implementation of the switching element of the Fairisle ATM switch. This verification was performed using the HOL90 theorem proving system so is fully machine-checked. We give here all the definitions used in the verification together with the main correctness theorems proved. Fairisle switches are in use in a working network, switching real data. Thus, this work constitutes a realistic formal verification case study.

Contents

1 Introduction

In this report we describe in detail the formal verification of the Fairisle 4 by 4 switching element. This verification was performed using the HOL90 theorem proving system so is fully machine-checked.

This report should be read in conjunction with the companion report [1]. It gives an overview of the project and briefly describes the switching element. It also provides a tutorial for the specification and verification techniques used.

In Section 2 we give general definitions upon which the formal specifications of the hardware are based. We give all the underlying HOL definitions used other than those pre-defined in the HOL90 system. We then describe the verification of each module including the full switching fabric in Section 3. For each module in the design we give structural and behavioural specifications and give the correctness theorem proved about it. For comparison, we also give the original HDL descriptions from which the HOL structural description was derived. For all but the simplest modules we give a circuit diagram of the implementation. In the circuit diagrams, we use the notation `w[i]` to refer to bit `i` of word `w`; `w[[i]]` to refer to the word consisting of the `i`th bits of each of the sub-words of word `w`; `w[i,j]` to refer to word with bits the `i`th and `j`th bits of word `w`; and `w[-i]` to refer to the word `w` with the `i`th bit omitted.

The top-level description of the switching element can be found in Section 3.43. The behavioural specification depends on the general purpose functions in Section 2.9. Its structural specification relies on those of all the other modules and the specifications of the basic constructs (such as logic gates) used. The latter are described in Section 2.8.

A detailed account of the HOL system and the higher order logic notation is given by Gordon and Melham [2]. A description of the word library is given by Wong [3]. We do not give type information in the definitions. This can normally be determined from context. Normally, the most general types possible are used.

2 General Definitions

In this section we give the general definitions upon which the specifications of the modules are based. For example, we give the specifications for the logic gates upon which the descriptions of the implementations depend. We also give definitions about round-robin arbitration, upon which the behavioural specifications are based. Many of the definitions given here would be of direct use in other formal verification projects.

2.1 Words

In this section we describe additional definitions about words that extend those given in the word library. Most concern multi-level words.

2.1.1 REMBIT

The arguments to `REMBIT` are a bit position, `k`, indexing from the left and from zero and a word, `w`. It returns the word with the indexed bit removed, ie. a word one bit shorter. This is achieved by concatenating the initial and final segments, excluding the chosen bit, together. The bit position specified must be less than the size of the word `w`.

```
⊢ ∀ n.
    ∀ w ::(PWORDLEN n).
      ∀ k.
        k < n ⊃
        (REMBIT k w = WCAT (WSEG (PRE (n - k)) (SUC k) w,WSEG k 0 w))
```

2.1.2 BITS

`BITS` extracts the nth bit from each subword [1] of a given word, producing a new word with the same length as the original, but one level shallower. It is equivalent to applying `BIT` to each element of the word. In the definition we do not explicitly give the word argument, since it is the same on both sides of the definition. Both `(BITS n)` and `(WMAP (BIT n))` are functions which when given a word, return a word. `WMAP` is a word map function. It applies the given function (here `(BIT n)`) to each element of a word.

```
⊢ ∀ n. BITS n = WMAP (BIT n)
```

2.1.3 WSEGS

`WSEGS` extracts the specified segment from each subword of a given word, producing a new word with the same length as the original, but with the inner words shorter. It is equivalent to applying `WSEG` to each element of the word.

```
⊢ ∀ n m. WSEGS n m = WMAP (WSEG n m)
```

2.1.4 BWORDLEN

`BWORDLEN` returns the length of the sub-words of a word. Since we constrain all bits of a word of words to be the same length, the use of `(BIT 0)` in the definition is arbitrary.

```
⊢ ∀ w. BWORDLEN w = WORDLEN (BIT 0 w)
```

[1] A subword is a bit of a two level word

2.1.5 PBITLEN

`PBITLEN` tests that the subwords of a word are of a given length. The bits at all positions `k` up to the word length should be of the given length. `PBITLEN` is used to constrain the bits of words of words to all be the same length.

```
⊢ ∀ n w. PBITLEN n w = (∀ k. k < WORDLEN w ⊃ PWORDLEN n (BIT k w))
```

2.1.6 PWORD2LEN

`PWORD2LEN` tests for the size of a word and its subwords for given lengths.

```
⊢ ∀ n m w. PWORD2LEN n m w = PWORDLEN n w ∧ PBITLEN m w
```

2.1.7 MKW

`MKW` creates a word from a function from bit positions (counting from zero) to values. It takes as argument the size of word to be created and the generating function. The result of applying the function to each bit position is concatenated on to the result generated by applying `MKW` to the remainder of the positions.

```
⊢ (∀ f. MKW 0 f = WORD []) ∧ (∀ n f. MKW (SUC n) f = WCAT (WORD [f n],MKW n f))
```

2.1.8 MKCW

`MKCW` creates a word of a given size with each element identical to the given value. The generating function `(λi. v)` passed to `MKW` maps all bit positions `i` to the value `v`.

```
⊢ ∀ n v. MKCW n v = MKW n (λ i. v)
```

2.1.9 MKW2

`MKW2` creates a shuffled word from that produced by a given generating function. The generating function creates a word of words. This is converted into a new word where each bit position holds a word of the values at that bit position in each of the original subwords.

```
⊢ ∀ n m f. MKW2 n m f = MKW n (λ i. BITS i (MKW m f))
```

2.1.10 WRFOLD_DEF

WRFOLD does a fold operation over words. It can be used with operations that do not have an identity, as it is not defined for an empty list. For lists of length 1, the single element results. For other lengths, the result is equivalent to applying the operation cumulatively to each of the elements and the previous result. For example, if applied to a word WORD[b_1;b_2;b_3] with operator op, it is equivalent to b_1 op (b_2 op b_3).

```
⊢ ∀ f b w.
    WRFOLD f (WCAT (WORD [b],w)) = ((PWORDLEN 0 w) ⇒ b |  (f b (WRFOLD f w)))
```

2.1.11 ZEROW

ZEROW returns a word of a given size where each bit has value F. Such a word is the binary equivalent of 0. It is therefore defined in terms of the function NBWORD which converts a number to a binary encoded word.

```
⊢ ∀ n. ZEROW n = NBWORD n 0
```

2.2 Position Vectors

In this section we define position vectors: natural numbers encoded as unary boolean words. Exactly one bit is true in a position vector and its position indicates the number represented.

2.2.1 NPVEC

NPVEC converts a natural number, n, to a position vector: a boolean word which has value T in the position indicated by the natural number, with all other positions holding value F. The generating function ($= n) passed to MKW tests if each bit is equal to n. A $ before an infix operator such as = suppresses its infix status, allowing it to be curried.

```
⊢ ∀ wordsize n. NPVEC wordsize n = MKW wordsize ($= n)
```

2.2.2 PVECN

PVECN converts a position vector to a natural number. The position of the first true bit in the word is returned. It is not defined for words of length zero. Each bit is tested in turn. If it is true, the result is the length of the remainder of the word.

```
⊢ ∀ b w. PVECN (WCAT (WORD [b],w)) = (b ⇒ (WORDLEN w) |  (PVECN w))
```

2.2.3 WPVEC

WPVEC converts a word to a position vector. The word is taken as representing a position. A word of the given size is produced with value F at all positions except that indicated by the word argument where it is true. It is equivalent to converting a word to a number and then converting that to a position vector.

```
⊢ ∀ wordsize w. WPVEC wordsize w = NPVEC wordsize (BNVAL w)
```

2.3 Signals

In this section we describe operators on signals from time to words. They are lifted versions of word operators.

2.3.1 SLIFT

SLIFT converts operators on words to operators on signals: functions from time to words. Given a function f and signal sig, for any time t, it applies the signal to the time to give a word and then applies f to the result.

```
⊢ ∀ f sig. SLIFT f sig = (λ t. f (sig t))
```

2.3.2 SWORD

SWORD converts a word of signals to a signal of a word. At a time t the result is a word with each bit the same as the corresponding bit in the original word at time t.

```
⊢ ∀ l. SWORD l = (λ t. WORD (MAP (λ f. f t) l))
```

2.3.3 SBIT

SBIT accesses the nth bit of the word at a given time on the signal. It is a lifted version of BIT.

```
⊢ ∀ n. SBIT n = SLIFT (BIT n)
```

2.3.4 SWSEG

SWSEG accesses a segment of the word at a given time on the signal. It is a lifted version of WSEG.

```
⊢ ∀ n m. SWSEG n m = SLIFT (WSEG n m)
```

2.3.5 SWSEGS

`SWSEGS` accesses the nth bits of each of the subwords within the word at a given time on the signal. It is a lifted version of `WSEGS`.

```
⊢ ∀ n. SBITS n = SLIFT (BITS n)
```

2.3.6 SBITS

`SBITS` accesses the nth bits of each of the subwords within the word at a given time on the signal. It is a lifted version of `BITS`.

```
⊢ ∀ n. SBITS n = SLIFT (BITS n)
```

2.3.7 SREMBIT

`SREMBIT` applied to a signal at a given time gives the word value of the signal at that time with the specified bit omitted. It is a lifted version of `REMBIT`.

```
⊢ ∀ i. SREMBIT i = SLIFT (REMBIT i)
```

2.3.8 SEXISTSABIT

`SEXISTSABIT` returns a boolean signal. It is true at a given time if the predicate `P` holds of any bit at that time. It is a lifted version of `EXISTSABIT`.

```
⊢ ∀ P. SEXISTSABIT P = SLIFT (EXISTSABIT P)
```

2.3.9 SIGLEN

`SIGLEN` gives the length of the words of a signal. At all times, the signal should return a word of the same length. It is therefore arbitrarily defined in terms of the word at time 0.

```
⊢ ∀ sig. SIGLEN sig = WORDLEN (sig 0)
```

2.3.10 BSIGLEN

`BSIGLEN` gives the length of the subwords of a 2 level signal. At all times, the signal should return a word of the same length. It is therefore arbitrarily defined in terms of the word at time 0. Also all bits should be of the same length, so bit 0 is arbitrarily chosen.

```
⊢ ∀ sig. BSIGLEN sig = WORDLEN (BIT 0 (sig 0))
```

2.3.11 PSIGLEN

PSIGLEN tests for the length of the words of a signal. A signal is a given length if at all times its constituent words are that length.

```
⊢ ∀ n. PSIGLEN n = (λ sig. ∀ t. PWORDLEN n (sig t))
```

2.3.12 PSIG2LEN

PSIG2LEN tests for the length of the words and their subwords of a signal. A signal is a given size if at all times its constituent words are that size.

```
⊢ ∀ n m. PSIG2LEN n m = (λ sig. ∀ t. PWORD2LEN n m (sig t))
```

2.4 General Definitions

In this section we define some general functions concerning ranges of natural numbers, and error results.

2.4.1 BETWEEN

BETWEEN states that the value i lies in the range between m and n, inclusive of m but not of n.

```
⊢ ∀ m i n. BETWEEN m i n = m <= i ∧ i < n
```

2.4.2 TO

TO is a curried version of "greater than". It is used in structural hardware descriptions to indicate the number of occurrences of a replicated piece of hardware.

```
⊢ TO = $>
```

2.4.3 DUP

DUP states that for value i in the range between m and n, inclusive of m but not of n, the predicate P holds. It is used in structural hardware specifications to duplicate occurrences of a piece of hardware specified by the predicate and parameterised by an instance number. It is also used in temporal specifications to specify that a predicate holds over some time range.

```
⊢ ∀ m n P. DUP m n P = (∀ i. BETWEEN m i n ⊃ P i)
```

2.4.4 Results

A type for indicating good results and error results is defined. The type is polymorphic so can be used to return results of any type.

```
Result = NO_RESULT |  RESULT of 'a
```

`ResultOf` returns the value of a non-error result.

```
⊢ ∀ a. ResultOf (RESULT a) = a
```

2.5 Signal values over intervals

In this section we define some general functions concerning the values of signals (functions from time to values of some type) over time intervals.

2.5.1 DURING

`DURING` specifies the values on a signal `sig` during some interval, inclusive of the initial time `ts`, but not of the end time `te`. The values are specified by a function `f`. At any time `t` in the interval, the value of the signal should equal the value of the function at that time. Nothing is specified outside the interval.

```
⊢ ∀ ts te sig f. DURING ts te sig f = DUP ts te (λ t. sig t = f t)
```

2.5.2 STABLE

`STABLE` specifies that the value `value` on a signal `sig` is stable during some interval, inclusive of the initial time `ts`, but not of the end time `te`.

```
⊢ ∀ ts te sig value. STABLE ts te sig value = DURING ts te sig (λ t. value)
```

2.5.3 NEXT

`NEXT` specifies that `t2` is the next time after `t1` that boolean signal `sig` is true. `t2` should be greater than `t1`. The signal should be true at `t2` and false from then until the end of the interval.

```
⊢ ∀ t1 t2 sig.
     NEXT t1 t2 sig = t1 < t2 ∧ sig t2 ∧ DUP (t1 + 1) t2 (λ t. ~ (sig t))
```

2.6 Frames

In this section we give definitions of time frames of various kinds. In its simplest form a frame is just a period of time bounded by consecutive times a signal is true. For the switching element they are used to define the period over which a cell is processed. More complex definitions are used to include information about when the cells arrive within the frame. Different definitions are required to define the behaviour of different modules because the signals involved are delayed by different amounts before they arrive at the boundary of the module.

2.6.1 FRAME

`FRAME` defines a simple frame. It specifies a period of time over which adjacent occurrences of a given signal are true. Start and end times, `ts` and `te` mark the boundaries of a frame on a given boolean signal `sig` if the signal is true at time `ts`, and the next time it is true is at time `te`.

```
⊢ ∀ ts te sig. FRAME ts te sig = sig ts ∧ NEXT ts te sig
```

2.6.2 Inactive Frames

`IFRAME` and `IFRAME1` define inactive frames; that is, frames in which no active signals arrive within the frame. Start and end times, `ts` and `te` mark the boundaries of a frame on a signal `sig`. Throughout this period the active signal remains low. `IFRAME` and `IFRAME1` give two different flavours of this basic definition. For both, the frame start signal into the fabric defines the boundaries of the frame.

`IFRAME` is used to specify the behaviour of the whole switching element. Here the active signal contains boolean words with one bit corresponding to the active bits of cells from each input port. All bits in this word must remain low throughout the cycle including at time `ts` but not `te`. This is expressed using `STABLE`. The expression `(SEXISTSABIT I active)` represents a signal which is high whenever a bit of signal `active` is set high.

`IFRAME1` is used to specify the behaviour of the arbiter. Here the active signal is a single boolean signal corresponding to the active bit of the cell from a single input port. This signal is delayed by one cycle more than the frame start signal in the path from the outside of the element. Therefore, the active signal must be low from time `ts+1` to `te+1`.

```
⊢ ∀ ts te sig active.
    IFRAME ts te sig active =
    FRAME ts te sig ∧ STABLE ts te (SEXISTSABIT I active) F
```

```
⊢ ∀ ts te sig active.
    IFRAME1 ts te sig active =
    FRAME ts te sig ∧ STABLE (ts + 1) (te + 1) active F
```

2.6.3 Active Frames

`AFRAME`, `AFRAME1` and `AFRAME2` define active frames. An active frame is one in which an active signal does arrive; that is a given boolean word signal remains low throughout the frame.

Start and end times, `ts` and `te` mark the boundaries of a frame on a signal `sig`. A further time `ta` indicates the first time within the frame that the active signal is high. `AFRAME`, `AFRAME1` and `AFRAME2` give different flavours of this basic definition. For each, the frame start signal into the fabric defines the boundaries of the frame.

`AFRAME` is used to specify the behaviour of the whole switching element. Here the active signal contains boolean words with one bit corresponding to the active bits of cells from each input port. All bits in this word must remain low within the frame until the active time. At least one must go high at the active time. The active time must be greater than or equal to the start time, and must be at least two cycles prior to the end of the frame.

`AFRAME1` is used to specify the behaviour of the arbiter. Here the active signal is a single boolean signal corresponding to the active bit of the cell from a single input port. Time `ta` must be the next time after the start time at which the active signal goes high. It must be at least two cycles prior to the end of the frame. The difference in timing is due to the different delays experienced by the active and frame start signals before reaching the arbiter.

`AFRAME2` is used to specify the behaviour of the fabric element less its output buffers and latches. The difference between it and `AFRAME` is just in the range of times that the active signal can occur, due to the delay caused by the latch on the data signal.

```
⊢ ∀ ts ta te sig active.
    AFRAME ts ta te sig active =
    FRAME ts te sig ∧
    STABLE ts ta (SEXISTSABIT I active) F ∧
    ts <= ta ∧
    SEXISTSABIT I active ta ∧
    ta + 1 < te
```

```
⊢ ∀ ts ta te sig active.
    AFRAME1 ts ta te sig active =
    FRAME ts te sig ∧ NEXT ts ta active ∧ ta < te
```

```
⊢ ∀ ts ta te sig active.
    AFRAME2 ts ta te sig active =
    FRAME ts te sig ∧
    STABLE ts ta (SEXISTSABIT I active) F ∧
    ts < ta ∧
    SEXISTSABIT I active ta ∧
    ta < te
```

2.7 Logical Connectives

In this section we define some logical connectives. They are used later to describe the behaviour of the Xilinx logic gates and latches.

2.7.1 Jk

Jk describes the action of a JK flip-flop. If the value stored in the flip-flop old is true then the next state is the negated k input. Otherwise it is the value on the j input.

```
⊢ ∀ j k old. Jk j k old = (old ⇒ (∼ k) |  j)
```

2.7.2 JkE

JkE describes the action of a JK flip-flop with an enable input. If the enable bit is set it behaves as a normal JK flip-flop. Otherwise, it retains its state.

```
⊢ ∀ j k old enable. JkE j k old enable = (enable ⇒ (Jk j k old) |  old)
```

2.7.3 XOR

XOR is the boolean exclusive-OR function.

```
⊢ ∀ a b. a XOR b = a ∧ ∼ b ∨ b ∧ ∼ a
```

2.7.4 XNOR

XNOR is the boolean exclusive-NOR function

```
⊢ ∀ a b. a XNOR b = a = b
```

2.7.5 LGATE

LGATE specifies a logic gate for a simple given binary logic operation. It only works for words of size greater than 2 if the operation is applied pair wise (e.g., AND and OR but not NAND or NOR). The value on the output at any time should be the folded value on the input at that time.

```
⊢ ∀ f xI xO. LGATE f (xI,xO) = xO = (λ t. WRFOLD f (xI t))
```

2.7.6 NLGATE

NLGATE specifies a logic gate for a binary logic operation, which first performs the operation over the word and then negates the result (e.g., for NAND or NOR).

```
⊢ ∀ f xI xO. NLGATE f (xI,xO) = xO = (λ t. ∼ (WRFOLD f (xI t)))
```

2.7.7 DEL1

DEL1 delays a signal for one time unit.

```
⊢ ∀ ins outs. DEL1 (ins,outs) = (∀ t. outs (t + 1) = ins t)
```

2.7.8 WIRE

WIRE is a delayless wire. It has no effect.

```
⊢ ∀ xI xO. WIRE (xI,xO) = (∀ t. xO t = xI t)
```

2.8 Xilinx Logic Gates and Latches

In this section we define the behaviour of the Xilinx logic gates and basic macros used in the HDL description of the switching fabric. They are primitive objects of structural specifications. Only a behavioural specification is given, not an implementation. We do not verify these primitives. We assume that these descriptions capture their behaviour sufficiently accurately. Logic gates are specified to have zero delay. Thus problems due to delays on the gates will not be highlighted in proofs based on these definitions. For our purposes this was considered satisfactory.

We give both the HDL and HOL definitions for the primitives. For most primitives the HDL version consists of only a stub with no body. This is because they are also primitives of the simulator. Their definitions are defined by the simulator.

In the HDL, a separate definition is given for a logic gate for each possible number of inputs. We use words which allows us to give a generic definition of each family.

Xilinx latches power up in a reset state. We have not included this in the specifications of the latches. Nothing is stated (so nothing can be proved) about the initial values in the latches. The behaviour of the switching element is not dependent on the initial values of these registers so the information was not needed in the proofs. By omitting the information, the specifications and proofs are also applicable to implementations using technology that does not reset latches on power-up.

2.8.1 LOCAL

`LOCAL` is used in structural hardware descriptions to declare local variables. It is just existential quantification re-named to make it more readable. Local variables contain values that are hidden from the outside world. Existential quantification expresses this by saying that all we know about the value is that it exists.

HOL

```
⊢ $LOCAL = (λ P. $∃  P)
```

2.8.2 FOR

`FOR` is used in structural hardware descriptions to declare local variables. It is just existential quantification re-named to make it more readable. Local variables contain values that are hidden from the outside world. existential quantification expresses this by saying that all we know about the value is that it exists.

HOL

```
⊢ $FOR = (λ P. $∀  P)
```

2.8.3 XiINV

`XiINV` describes the Xilinx inverter.

HOL

```
⊢ ∀ xI xO. XiINV (xI,xO) = (∀ t. xO t = ~ (xI t))
```

HDL

```
DEF XiINV (xI: IN; xO: IO);
BEGIN
END;
```

2.8.4 XiAND

`XiAND` describes the family of Xilinx AND gates having various numbers of inputs.

HOL

```
⊢ ∀ xI xO. XiAND (xI,xO) = LGATE $∧ (xI,xO)
```

HDL

```
DEF XiAND2 (x1, x2: IN; xO: IO);
BEGIN
END;

DEF XiAND3 (x1, x2, x3: IN; xO: IO);
BEGIN
END;

DEF XiAND4 (x1, x2, x3, x4: IN; xO: IO);
BEGIN
END;

DEF XiAND5 (x1, x2, x3, x4, x5: IN; xO: IO);
BEGIN
END;
```

2.8.5 `XiNAND`

`XiNAND` describes the family of Xilinx NAND gates having various numbers of inputs.

HOL

```
⊢ ∀ xI xO. XiNAND (xI,xO) = NLGATE $∧ (xI,xO)
```

HDL

```
DEF XiNAND2 (x1, x2: IN; xO: IO);
BEGIN
END;

DEF XiNAND3 (x1, x2, x3: IN; xO: IO);
BEGIN
END;

DEF XiNAND4 (x1, x2, x3, x4: IN; xO: IO);
BEGIN
END;

DEF XiNAND5 (x1, x2, x3, x4, x5: IN; xO: IO);
BEGIN
END;
```

2.8.6 `XiOR`

`XiOR` describes the family of Xilinx OR gates having various numbers of inputs.

HOL

```
⊢ ∀ xI xO. XiOR (xI,xO) = LGATE $∨ (xI,xO)
```

HDL

```
DEF XiOR2 (x1, x2: IN; xO: IO);
BEGIN
END;

DEF XiOR3 (x1, x2, x3: IN; xO: IO);
BEGIN
END;

DEF XiOR4 (x1, x2, x3, x4: IN; xO: IO);
BEGIN
END;

DEF XiOR5 (x1, x2, x3, x4, x5: IN; xO: IO);
BEGIN
END;

DEF XiOR6 (x1, x2, x3, x4, x5, x6: IN; xO: IO);
BEGIN
END;
```

2.8.7 XiNOR

XiNOR describes the family of Xilinx NOR gates having various numbers of inputs.

HOL

```
⊢ ∀ xI xO. XiNOR (xI,xO) = NLGATE $∨ (xI,xO)
```

HDL

```
DEF XiNOR2 (x1, x2: IN; xO: IO);
BEGIN
END;

DEF XiNOR3 (x1, x2, x3: IN; xO: IO);
BEGIN
END;

DEF XiNOR4 (x1, x2, x3, x4: IN; xO: IO);
BEGIN
END;

DEF XiNOR5 (x1, x2, x3, x4, x5: IN; xO: IO);
BEGIN
END;
```

2.8.8 `XiXOR2`

`XiXOR2` describes a Xilinx XOR gate having 2 inputs. If there are more than 2 inputs it performs a cumulative pairwise XOR, rather than a true XOR.

HOL

```
⊢ ∀ xI xO. XiXOR2 (xI,xO) = LGATE $XOR (xI,xO)
```

HDL

```
DEF XiXOR2 (x1, x2: IN; xO: IO);
BEGIN
END;
```

2.8.9 `XiXNOR2`

`XiXNOR2` describes a Xilinx XNOR gate having 2 inputs. If there are more than 2 inputs it performs a cumulative pairwise XNOR, rather than a true XNOR.

HOL

```
⊢ ∀ xI xO. XiXNOR2 (xI,xO) = LGATE $XNOR (xI,xO)
```

HDL

```
DEF XiXNOR2 (x1, x2: IN; xO: IO);
BEGIN
END;
```

2.8.10 `XiIBUF`

`XiIBUF` joins an external wire to an internal one with no delay.

HOL

```
⊢ ∀ inpEXT inp. XiIBUF (inpEXT,inp) = WIRE (inpEXT,inp)
```

HDL

```
DEF XiIBUF (xI: IN; xO: IO);
BEGIN
END;
```

2.8.11 `XiOBUF`

`XiOBUF` joins an external wire to an internal one with no delay.

HOL

```
⊢ ∀ outEXT out. XiOBUF (outEXT,out) = WIRE (outEXT,out)
```

HDL

```
DEF XiOBUF (xI: IN; xO: IO);
BEGIN
END;
```

2.8.12 XiOUTFFd

`XiOUTFFd` describes the Xilinx output latch. It gives a single cycle delay.

HOL

```
⊢ ∀ xI xO. XiOUTFFd (xI,xO) = DEL1 (xI,xO)
```

HDL

```
DEF XiOUTFFd (xD, xC: IN; xQ: IO);
BEGIN
END;
```

2.8.13 XiINFFd

`XiINFFd` describes the Xilinx input latch. It gives a single cycle delay.

HOL

```
⊢ ∀ xI xO. XiINFFd (xI,xO) = DEL1 (xI,xO)
```

HDL

```
DEF XiINFFd (xD, xC: IN; xQ: IO);
BEGIN
END;
```

2.8.14 XiDFFd

`XiDFFd` describes the Xilinx latch. It gives a single cycle delay.

HOL

```
⊢ ∀ xI xO. XiDFFd (xI,xO) = DEL1 (xI,xO)
```

HDL

```
DEF XiDFFd (xD, xC: IN; xQ: IO);
BEGIN
END;
```

2.8.15 XiDFFrd

XiDFFrd describes the Xilinx latch with reset data line. It gives a single cycle delay if the rd input is low and otherwise it outputs low.

HOL

```
⊢ ∀ xI rd xO. XiDFFrd ((xI,rd),xO) = (∀ t. xO (t + 1) = ((rd t) ⇒ F |  (xI t)))
```

HDL

```
(* Constrained version of XiDFF with reset data *)
DEF XiDFFrd (xD, xC, xRD: IN; xQ: IO);
BEGIN
END;
```

2.8.16 JKFF

JKFF describes the JK flip-flop. We have taken it as a basic component, though it is given an implementation in the HDL.

HOL

```
⊢ ∀ j k q qBar.
     JKFF ((j,k),q,qBar) =
     (∀ t. (q (t + 1) = Jk (j t) (k t) (q t)) ∧ (qBar t = q t))
```

HDL

```
DEF JKFF (j, k, c: IN; q, qBar: IO);

d, dOne, okOne, dZeroBar, jBar, kBar : IO;

BEGIN

    Jkff := XiDFFd (d, c, q);
    OrD := XiOR2 (dOne, okOne, d);
    AndOKOne := XiAND2 (dZeroBar, q, okOne);
    AODOne := AO (j, kBar, j, qBar, dOne);
    AOIZeroBar := AOI (k, jBar, k, q, dZeroBar);
    InvQ := XiINV (q, qBar);
    InvK := XiINV (k, kBar);
    InvJ := XiINV (j, jBar);
END;
```

2.8.17 JKFFce

`JKFFce` describes the JK flip-flop with `enable` input. If the `enable` bit is set it behaves as a normal JK flip-flop. Otherwise, it retains its state. We have taken it as a basic component, though it is given an implementation in the HDL.

HOL

```
⊢ ∀ j k ce q.
     JKFFce ((j,k,ce),q) = (∀ t. q (t + 1) = JkE (j t) (k t) (q t) (ce t))
```

HDL

```
DEF JKFFce (j, k, c, ce: IN; q: IO);

d, dOne, okOne, dZeroBar, qBar, jBar, kBar : IO;

BEGIN
    Jkffce := XiDFFce (d, c, ce, q);
    OrD := XiOR2 (dOne, okOne, d);
    ANDOkOne := XiAND2 (dZeroBar, q, okOne);
    AODOne := AO (j, kBar, j, qBar, dOne);
    AOIZeroBar := AOI (k, jBar, k, q, dZeroBar);
    InvQ := XiINV (q, qBar);
    InvK := XiINV (k, kBar);
    InvJ := XiINV (j, jBar);
END;
```

2.8.18 AO

`AO` ANDs its two pairs of inputs and ORs the results. We have taken it as a basic component, though it is given an implementation in the HDL.

HOL

```
⊢ ∀ a b c d x0. AO ((a,b,c,d),x0) = (∀ t. x0 t = a t ∧ b t ∨ c t ∧ d t)
```

HDL

```
DEF AO (a, b, c, d: IN; x: IO);

s, t: IO

BEGIN
    And1 := XiAND2 (a, b, s);
    And2 := XiAND2 (c, d, t);
    Or  := XiOR2 (s, t, x);
END;
```

2.9 Round Robin Arbitration and Priority Filtering

In this section we give the underlying definitions for round robin switching with priority and acknowledgement. These form the basis of the behavioural specifications of the switching element and of many of its constituent modules. We define: round robin arbitration; the process of filtering out the highest priority requests; a function which picks the successful input for a given output; and functions which specify the new round robin last successful value, the new data to be output and the acknowledgement signal to be sent. We are not concerned in this section with when these processes occur within a frame, only of their function.

2.9.1 `DecoderRequests`

`DecoderRequests` is called once for each input port. It returns a position vector with one element for each output port, indicating whether this input port was making a request (of any priority) for that output port. If the input port is not making a request, all positions will be F. The request `req` is encoded as a binary word. If no request is being made, the active signal `act` is false.

```
⊢ ∀ act req. DecoderRequests act req = (act ⇒ (WPVEC 4 req) |  (ZEROW 4))
```

2.9.2 `DecoderPriorities`

`DecoderPriorities` is called once for each input port. It returns a position vector with one element for each output port, indicating whether this input port was making a high priority request for that output port. If the input port is not making a priority request, all positions will be F. A high priority request is indicated by both the active, `act`, and priority, `pri` bits being high.

```
⊢ ∀ act pri req.
     DecoderPriorities act pri req =
     ((act ∧ pri) ⇒ (WPVEC 4 req) |  (ZEROW 4))
```

2.9.3 `SUC_MODN`

This function defines the wrap around for Round Robin arbitration. It is the successor function modulo n, where n is one greater than the highest possible arbitration result.

```
⊢ ∀ n last. SUC_MODN n last = ((last = n) ⇒ 0 |  (SUC last))
```

2.9.4 `RoundRobinArbiter`

Given an indication of the last value selected, round robin arbitration returns the next highest value requested, with suitable wrap around from the highest possible request to the lowest.

The round robin arbiter takes a set of natural number requests and a natural number indicating the last successful request. It returns the new successful request or an error if no requests were made (i.e., the request set is empty). `RoundRobin` is called. It tries successively higher values above the last successful request until it finds a request that is in the set of requests. It is defined in terms of a counter that ensures that the function does terminate. Provided the counter is initially the value of the highest request possible and the request set is not empty, it will not terminate before a result is obtained.

We arbitrarily return 0 if n is 0. This should never arise, but makes part of the proof simpler.

```
⊢ (∀ request_set last. RoundRobin 0 request_set last = 0) ∧
   (∀ n request_set last.
     RoundRobin (SUC n) request_set last =
     (let trynext = SUC_MODN 3 last
      in
      ((trynext IN request_set)
       ⇒ trynext
       |  (RoundRobin n request_set trynext))))
```

```
⊢ ∀ n request_set last.
     RoundRobinArbiter n request_set last =
     ((request_set = {})
      ⇒ NO_RESULT
      |  (RESULT (RoundRobin n request_set last)))
```

2.9.5 PriorityRequests

The function `PriorityRequests` filters out low priority requests for an output if there is a high priority request for it. It also filters out the request signals from non-active inputs. It takes three boolean words and returns a boolean word result. Each word contains one bit for each input channel. Each bit is a flag indicating whether the property described by the word is true for that input. The first word indicates whether the input is actively making a request. The second indicates which inputs are high priority ones. The third indicates which are making requests for the output under consideration. If there are any active, high priority requests for the output, then the result indicates which they were. Otherwise the result indicates the low priority active requests for the output.

```
⊢ ∀ actives priorities requests.
     PriorityRequests actives priorities requests =
     (let high_priority_requests = priorities WAND actives WAND requests
      in
      (let general_requests = actives WAND requests
       in
       ((EXISTSABIT I high_priority_requests)
        ⇒ high_priority_requests
        |  general_requests)))
```

2.9.6 RequestsToArbitrate

`RequestsToArbitrate` returns a set of requests given a boolean word with each bit indicating whether the corresponding input is making a request. It scans each bit in turn. If it holds the value `T`, then its position is added to the request set.

```
⊢ (∀ requests. RequestsToArbitrate 0 requests = {}) ∧
   (∀ n requests.
     RequestsToArbitrate (SUC n) requests =
     ((BIT n requests)
      ⇒ (n INSERT RequestsToArbitrate n requests)
      | (RequestsToArbitrate n requests)))
```

2.9.7 SuccessfulInput

`SuccessfulInput` chooses on a Round Robin basis the successful input, if any, for a single output port. It takes as arguments a boolean word `req`, which indicates which input ports are making requests for which output ports. It has one bit per input port. A value `T` in a bit position indicates that the input port corresponding to that position is requesting the output of interest. It also takes a natural number indicating the last successful input for this output. If there are no requests for this output then a `NO_RESULT` value is returned. Otherwise the result is that given by performing Round Robin arbitration on the set of requests.

```
⊢ ∀ last req.
     SuccessfulInput last req =
     (let request_set = RequestsToArbitrate (WORDLEN req) req
      in
      (RoundRobinArbiter (WORDLEN req) request_set last))
```

2.9.8 PickSuccessfulInput

`PickSuccessfulInput` chooses on a Round Robin basis the successful input, if any, for a given output port. It takes as arguments a boolean word indicating which inputs are active, one which indicates which inputs are currently high priority, and one indicating which inputs are requesting the output. It also takes a natural number indicating the last successful input for the output. It first creates a set of the highest priority active requests for the output. If there are no requests then a `NO_RESULT` value is returned. Otherwise the result is that given by performing Round Robin arbitration on the set of requests.

```
⊢ ∀ actives priorities requests last.
     PickSuccessfulInput actives priorities requests last =
     SuccessfulInput last (PriorityRequests actives priorities requests)
```

2.9.9 RequestsForPort

`RequestsForPort` converts a word of requested outputs (natural numbers) to a word indicating for each position whether it was a request for a given output.

```
⊢ ∀ port requests. RequestsForPort port requests = WMAP ($= port) requests
```

2.9.10 ChooseSuccessfulInput

`ChooseSuccessfulInput` chooses on a Round Robin basis the successful input, if any, for a given output port.

It takes boolean words giving the active and priority status of each input port, `actives` and `priorities` respectively; a natural number word indicating which output port each input port is requesting, `requests`; a natural number word giving the last successful input ports for the requested output port, `last`; and the number of the output port of interest `outportno`.

The inputs requesting this output are determined using `RequestsForPort`. These values are used to choose a successful input, with `PickSuccessfulInput`. If no input was requesting the output port a "no result" is returned.

```
⊢ ∀ actives priorities requests last outportno.
     ChooseSuccessfulInput actives priorities requests last outportno =
     (let requests_for_port = RequestsForPort outportno requests
      in
      (PickSuccessfulInput actives priorities requests_for_port last))
```

2.9.11 NACK

`NACK` is the negative acknowledgement signal. It is low.

```
⊢ NACK = F
```

2.9.12 FabricLast

`FabricLast` returns the new successful input after round robin arbitration for a given output port. It takes boolean words giving the active and priority status of each input port, `actives` and `priorities` respectively; a natural number word indicating which output port each input port is requesting, `requests`; a natural number word giving the last successful input ports for the output port, `last`; and the number of the output port `outportno`. If no input was requesting this output port, the last successful input port is unchanged, otherwise, it is the one chosen by round robin arbitration.

```
⊢ ∀ actives priorities requests last outportno.
    FabricLast actives priorities requests last outportno =
    (let successful_input =
             ChooseSuccessfulInput actives priorities requests last outportno
     in
     ((successful_input = NO_RESULT) ⇒ last |  (ResultOf successful_input)))
```

2.9.13 FabricDataOut

`FabricDataOut` returns the data to be output on a given output port. This will be the data input on the chosen input port. If no input is requesting the output port, then a default value is output.

`FabricDataOut` takes boolean words giving the active and priority status of each input port, `actives` and `priorities` respectively; a natural number word indicating which output port each input port is requesting, `requests`; a word of data – one data value for each input; a natural number word giving the last successful input port for this output port, `last`; the value of the default data, `default_data` and the number of the output port `outportno`.

A successful input port is chosen. If no input was requesting this output port, the default value is output. Otherwise, the data value input from the chosen input port is output.

```
⊢ ∀ actives priorities requests data_ins last default_data outportno.
    FabricDataOut actives priorities requests data_ins last default_data
      outportno =
    (let successful_input =
             ChooseSuccessfulInput actives priorities requests last outportno
     in
     ((successful_input = NO_RESULT)
      ⇒ default_data
      |  (BIT (ResultOf successful_input) data_ins)))
```

2.9.14 FabricAck

`FabricAck` returns the acknowledgement result for a given input port. If the input ports request was successful, then the result will be the acknowledgement from the requested output port. Otherwise, it will be a `NACK` signal.

`FabricAck` takes boolean words giving the active and priority status of each input port, `actives` and `priorities` respectively; a natural number word indicating which output port each input port is requesting, `requests`; a natural number word giving the last successful input port for each output port, `lasts`; a word of acknowledgements input – one acknowledgement for each output port – `ack_in`; and the number of the input port of interest `inportno`.

The output port requested by the input port under consideration is determined as is the last input granted for that output. The successful input port for the output port is then chosen. If the chosen input port is the one of interest, the acknowledgement from the requested output port results. If either a different input port or no input port was successful (indicating that the input port under consideration was not active), then a negative acknowledgement results.

```
⊢ ∀ actives priorities requests lasts ack_ins inportno.
     FabricAck actives priorities requests lasts ack_ins inportno =
     (let out_requested = BIT inportno requests
      in
      (let last = BIT out_requested lasts
       in
       (let successful_input =
                ChooseSuccessfulInput actives priorities requests last
                  out_requested
        in
        (let ack_in = BIT out_requested ack_ins
         in
         ((successful_input = RESULT inportno) ⇒ ack_in | NACK)))))
```

2.9.15 Accessing the fields of a header

`ActiveOfHeader` accesses the active bit (bit 0) of a header byte. `PriorityOfHeader` accesses the priority bit (bit 1) of a header byte. `RequestOfHeader` accesses the 2 route bits (bits 2 and 3) of a header byte.

```
⊢ ∀ header. ActiveOfHeader header = BIT 0 header
```

```
⊢ ∀ header. PriorityOfHeader header = BIT 1 header
```

```
⊢ ∀ header. RequestOfHeader header = BNVAL (WSEG 2 2 header)
```

2.9.16 Accessing the fields of a word of headers

`Actives` returns a word consisting of the active bits (bit 0) of each of the bytes in the given word of headers. `Priorities` returns a word consisting of the priority bits (bit 1) of each of the bytes in the given word of headers. `Requests` returns a word consisting of the route bits (bits 2 and 3) of each of the bytes in the given word of headers.

```
⊢ Actives = WMAP ActiveOfHeader
```

```
⊢ Priorities = WMAP PriorityOfHeader
```

```
⊢ Requests = WMAP RequestOfHeader
```

2.9.17 Fabric4x1Ack

Fabric4x1Ack specifies the acknowledgement to be sent to the port with port number inportno. It is given the header bytes headers, the previous grants made, lasts, and the acknowledgements from each output port, ack_ins. Round robin arbitration is performed for each output. If the output port that the input's header is requesting chooses the input port, then the acknowledgement from that output port is sent to the input port. Otherwise a negative acknowledgement is sent.

```
⊢ ∀ headers lasts ack_ins inportno.
     Fabric4x1Ack headers lasts ack_ins inportno =
     FabricAck (Actives headers) (Priorities headers) (Requests headers) lasts
       ack_ins
       inportno
```

2.9.18 Fabric4x1Last

Fabric4x1Last specifies the new most recent successful input port for the output port outportno. It is given the header bytes headers and the previous grant made last. Round robin arbitration is performed for each output to determine the new successful input ports.

```
⊢ ∀ headers last outportno.
     Fabric4x1Last headers last outportno =
     FabricLast (Actives headers) (Priorities headers) (Requests headers) last
       outportno
```

2.9.19 Fabric4x1DataOut

Fabric4x1DataOut specifies the data to be output for the output port outportno, given the default data to be output if it is not requested default_data_out. It is given the header bytes headers, the previous grant made last. Round robin arbitration is performed for each output to determine the new successful input ports.

```
⊢ ∀ default_data_out headers data_ins last outportno.
     Fabric4x1DataOut default_data_out headers data_ins last outportno =
     FabricDataOut (Actives headers) (Priorities headers) (Requests headers)
       data_ins
       last
       default_data_out
       outportno
```

3 The Modules

In this section we describe each of the modules used in the design of the switching element. We give an informal behavioural specification, followed by a formal HOL version. We then give the original HDL description of the implementation followed by the corresponding HOL definition.

Where the module is additional to that used in the HDL, we give the extract of HDL to which it corresponds. We highlight any major differences (other than surface syntax) in the descriptions. For example we highlight where multi-level words are used instead of a flat list of signal names. Finally we give the correctness theorems. For correctness theorems with complex proofs we give an informal overview. HOL definitions and theorems are enclosed in boxes.

3.1 IN_BUF

3.1.1 The Behavioural Specification

`IN_BUF` is a generic input buffer. It joins an external wire to an internal one with no delay.

```
⊢ ∀ inpEXT inp. IN_BUF_SPEC (inpEXT,inp) = WIRE (inpEXT,inp)
```

3.1.2 The Structural Specification

The HDL and HOL descriptions are basically the same, though the size is not fixed in the HOL definition. It is used for the `frameStart` and `ackIn` inputs.

```
⊢ ∀ inpEXT inp.
     IN_BUF (inpEXT,inp) =
     (FOR i ::(TO (SIGLEN inpEXT)). XiIBUF (SBIT i inpEXT,SBIT i inp))
```

Qudos HDL

```
FS := XiIBUF (frameStartEXT, frameStart);

Ai[0-3] := XiIBUF (ackInEXT[0-3], ackIn[0-3]);
```

3.1.3 The Correctness Statement

The correctness statement is generic. It states that input buffers of any given size `n` are correct. The proof is straightforward as are those for all the latches, and buffers. Most are done automatically by a tactic written to prove correctness theorems about hardware consisting of duplicated elements.

```
⊢ ∀ n.
     FOR inpEXT ::(PSIGLEN n).
       FOR inp ::(PSIGLEN n). IN_BUF (inpEXT,inp) ⊃ IN_BUF_SPEC (inpEXT,inp)
```

3.2 OUT_BUF

3.2.1 The Behavioural Specification

`OUT_BUF` is a generic output buffer. It joins an internal wire to an external one with no delay.

```
⊢ ∀ out outEXT. OUT_BUF_SPEC (out,outEXT) = WIRE (out,outEXT)
```

3.2.2 The Structural Specification

The HDL and HOL descriptions are basically the same, though the size is not fixed in the HOL definition. It is used for the `ackOut` output.

```
⊢ ∀ out outEXT.
    OUT_BUF (out,outEXT) =
    (FOR i ::(TO (SIGLEN out)). XiOBUF (SBIT i out,SBIT i outEXT))
```

Qudos HDL

```
Ao[0-3] := XiOBUF (ackOut[0-3], ackOutEXT[0-3]);
```

3.2.3 The Correctness Statement

The correctness statement is generic. It states that output buffers of any given size `n` are correct.

```
⊢ ∀ n.
    FOR out ::(PSIGLEN n).
      FOR outEXT ::(PSIGLEN n).
        OUT_BUF (out,outEXT) ⊃ OUT_BUF_SPEC (out,outEXT)
```

3.3 ILATCH

3.3.1 The Behavioural Specification

`ILATCH` is a generic input latch. It joins an external wire to an internal one with a single cycle delay.

```
⊢ ∀ inpEXT inp. ILATCH_SPEC (inpEXT,inp) = DEL1 (inpEXT,inp)
```

3.3.2 The Structural Specification

The main difference between the HDL and HOL descriptions is that the clock input is omitted as it is abstracted away from in the semantics. The size of the latter is not fixed in the HOL definition. It is given by the size of the arguments. It is used for each byte of the `d` input of the fabric.

```
⊢ ∀ inpEXT inp.
    ILATCH (inpEXT,inp) =
    (FOR i ::(TO (SIGLEN inpEXT)). XiINFFd (SBIT i inpEXT,SBIT i inp))
```

Qudos HDL

```
I[0-31] := XiINFFd (dEXT[0-31], clock, d[0-31]);
```

3.3.3 The Correctness Statement

The correctness statement is generic. It states that input latches of any given size `n` are correct.

```
⊢ ∀ n.
     FOR inpEXT ::(PSIGLEN n).
       FOR inp ::(PSIGLEN n). ILATCH (inpEXT,inp) ⊃ ILATCH_SPEC (inpEXT,inp)
```

3.4 IN_LATCH

3.4.1 The Behavioural Specification

`IN_LATCH` is a generic input latch for structured two level words. It joins an external wire to an internal one with a single cycle delay.

```
⊢ ∀ inpEXT inp. IN_LATCH_SPEC (inpEXT,inp) = DEL1 (inpEXT,inp)
```

3.4.2 The Structural Specification

The clock is omitted as it is abstracted away from in the semantics. The input and output are structured into two level words. Their size is not fixed in the definition. It is used for the `d` input of the switching element.

```
⊢ ∀ inpEXT inp.
     IN_LATCH (inpEXT,inp) =
     (FOR i ::(TO (SIGLEN inpEXT)). ILATCH (SBIT i inpEXT,SBIT i inp))
```

```
⊢ ∀ inpEXT inp.
     IN_LATCH_SIMPL (inpEXT,inp) =
     (FOR i ::(TO (SIGLEN inpEXT)). ILATCH_SPEC (SBIT i inpEXT,SBIT i inp))
```

Qudos HDL

```
I[0-31] := XiINFFd (dEXT[0-31], clock, d[0-31]);
```

3.4.3 The Correctness Statement

The correctness statement is generic. It states that input latches of any given size `n` are correct. The proof is straightforward. The implementation is built from non-primitive modules, so first a correctness theorem which refers to the behaviour of these components is proved. This is done automatically by a tactic written to prove correctness theorems about hardware consisting of duplicated elements.

```
⊢ ∀ n.
     FOR inpEXT ::(PSIGLEN n).
       FOR inp ::(PSIGLEN n).
         IN_LATCH_SIMPL (inpEXT,inp) ⊃ IN_LATCH_SPEC (inpEXT,inp)
```

It is then proved that a structural description based on the behaviour of the sub-modules implies one based on their structure. This is fairly simple to do using the correctness theorems about the sub-modules.

```
⊢ ∀ n.
     FOR inpEXT ::(PSIG2LEN n m).
       FOR inp ::(PSIG2LEN n m).
         IN_LATCH (inpEXT,inp) ⊃ IN_LATCH_SIMPL (inpEXT,inp)
```

It is trivial to prove the final correctness theorem from the above theorems.

```
⊢ ∀ n.
     FOR inpEXT ::(PSIG2LEN n m).
       FOR inp ::(PSIG2LEN n m).
         IN_LATCH (inpEXT,inp) ⊃ IN_LATCH_SPEC (inpEXT,inp)
```

3.5 OLATCH

3.5.1 The Behavioural Specification

`OLATCH` is a generic output latch. It joins an internal wire to an external one with a single cycle delay.

```
⊢ ∀ out outEXT. OLATCH_SPEC (out,outEXT) = DEL1 (out,outEXT)
```

3.5.2 The Structural Specification

The HDL and HOL descriptions are basically the same. The size is not fixed in the HOL definition and the clock argument is omitted. `OLATCH` is used for the `dOut` output.

```
⊢ ∀ out outEXT.
     OLATCH (out,outEXT) =
     (FOR i ::(TO (SIGLEN out)). XiOUTFFd (SBIT i out,SBIT i outEXT))
```

Qudos HDL

```
O[0-31] := XiOUTFFd (dOut[0-31], clock, dOutEXT[0-31]);
```

3.5.3 The Correctness Statement

The correctness statement is generic. It states that output latches of any given size `n` are correct.

```
⊢ ∀ n.
     FOR out ::(PSIGLEN n).
       FOR outEXT ::(PSIGLEN n).
         OLATCH (out,outEXT) ⊃ OLATCH_SPEC (out,outEXT)
```

3.6 OUT_LATCH

3.6.1 The Behavioural Specification

`OUT_LATCH` is a generic output latch for structured two level words. It joins an internal wire to an external one with a single cycle delay. The words involved are structured.

```
⊢ ∀ out outEXT. OUT_LATCH_SPEC (out,outEXT) = DEL1 (out,outEXT)
```

3.6.2 The Structural Specification

The clock is omitted as it is abstracted away from in the semantics. The input and output are structured into two level words. Their size is not fixed in the definition. `OUT_LATCH` is used for the `dOut` output.

```
⊢ ∀ out outEXT.
     OUT_LATCH (out,outEXT) =
     (FOR i ::(TO (SIGLEN out)). OLATCH (SBIT i out,SBIT i outEXT))
```

```
⊢ ∀ out outEXT.
     OUT_LATCH_SIMPL (out,outEXT) =
     (FOR i ::(TO (SIGLEN out)). OLATCH_SPEC (SBIT i out,SBIT i outEXT))
```

Qudos HDL

```
O[0-31] := XiOUTFFd (dOut[0-31], clock, dOutEXT[0-31]);
```

3.6.3 The Correctness Statement

The correctness statement is generic. It states that 2-level structured output latches are correct whatever their size.

```
⊢ ∀ n.
     FOR outpEXT ::(PSIG2LEN n m).
       FOR outp ::(PSIG2LEN n m).
         OUT_LATCH (outpEXT,outp) ⊃ OUT_LATCH_SPEC (outpEXT,outp)
```

3.7 LATCH

3.7.1 The Behavioural Specification

`LATCH` is a generic delay latch. It delays a signal for one time unit.

```
⊢ ∀ inp out. LATCH_SPEC (inp,out) = DEL1 (inp,out)
```

3.7.2 The Structural Specification

`LATCH` is an extra level of hierarchy over the HDL version. It is used several times in the design. For example, in the top level fabric design, it is used with size 8 for each byte in the delay on the input to the dataswitch and with size 4 for the delay on the input to the arbiter.

```
⊢ ∀ inp out.
     LATCH (inp,out) =
     (FOR i ::(TO (SIGLEN inp)). XiDFFd (SBIT i inp,SBIT i out))
```

Qudos HDL

```
Pause[0-31] := XiDFFd(d[0-31], clock, dPause[0-31]);

FFReq[0-15] := XiDFFd(req[0-15], clock, ltReq[0-15]);
```

3.7.3 The Correctness Statement

The correctness statement is generic. It states that input latches of any given size `n` are correct.

```
⊢ ∀ n.
     FOR inp ::(PSIGLEN n).
       FOR out ::(PSIGLEN n). LATCH (inp,out) ⊃ LATCH_SPEC (inp,out)
```

3.8 RLATCH

3.8.1 The Behavioural Specification

`RLATCH` is a generic delay latch with reset. It delays a signal for one time unit. However, if the signal `disable` is set, it outputs a zero word on the next cycle.

```
⊢ ∀ inp disable out.
     RLATCH_SPEC ((inp,disable),out) =
     (∀ t. out (t + 1) = ((disable t) ⇒ (ZEROW (SIGLEN out)) |  (inp t)))
```

3.8.2 The Structural Specification

RLATCH is an extra level of hierarchy over the HDL version. It is used in the multiplexor element of the dataswitch. Its size is not specified in the HOL definition. The clock signal is omitted from the HOL definition.

```
⊢ ∀ inp disable out.
     RLATCH ((inp,disable),out) =
     (FOR i ::(TO (SIGLEN inp)). XiDFFrd ((SBIT i inp,disable),SBIT i out))
```

Qudos HDL

```
BFF[0-1] := XiDFFrd (mux[0-1], clock, outputDisable, q[0-1]);
```

3.8.3 The Correctness Statement

The correctness statement is generic. It states that latches of any given size n are correct. The proof is straightforward. It is done largely automatically by the tactic written to prove correctness theorems about hardware consisting of duplicated elements. A small amount of additional work was needed to deal with the reset. This was trivial.

```
⊢ ∀ n disable.
     FOR inp ::(PSIGLEN n).
       FOR out ::(PSIGLEN n).
         RLATCH ((inp,disable),out) ⊃ RLATCH_SPEC ((inp,disable),out)
```

3.9 PAUSE

3.9.1 The Behavioural Specification

PAUSE is a generic delay latch for structured 2-level words; that is words consisting of words of bits. It delays a signal for one time unit.

```
⊢ ∀ d dPause. PAUSE_SPEC (d,dPause) = DEL1 (d,dPause)
```

3.9.2 The Structural Specification

PAUSE gives an extra level of hierarchy over the HDL version. It is used several times in the design. For example, in the top level fabric design, it is used once of size 4 by 8 for the delay on the input to the dataswitch and once of size 4 by 4 for the delay on the input to the arbiter. In HDL, the size must be fixed in the definition so each occurrence was given separately. In HOL the size does not need to be fixed, so a generic module can be defined. The clock signal is omitted from the HOL version.

```
⊢ ∀ d dPause.
     PAUSE (d,dPause) =
     (FOR i ::(TO (SIGLEN d)). LATCH (SBIT i d,SBIT i dPause))
```

```
⊢ ∀ d dPause.
     PAUSE_SIMPL (d,dPause) =
     (FOR i ::(TO (SIGLEN d)). LATCH_SPEC (SBIT i d,SBIT i dPause))
```

Qudos HDL

```
Pause[0-31] := XiDFFd(d[0-31], clock, dPause[0-31]);

FFReq[0-15] := XiDFFd(req[0-15], clock, ltReq[0-15]);
```

3.9.3 The Correctness Statement

The correctness statement is generic. It states that 2-level structured latches are correct whatever their size. The proof is straightforward. The implementation is built from non-primitive modules, so as with `IN_LATCH` the proof is split into three simple parts.

```
⊢ ∀ n.
     FOR d ::(PSIG2LEN n m).
       FOR dPause ::(PSIG2LEN n m). PAUSE (d,dPause) ⊃ PAUSE_SPEC (d,dPause)
```

3.10 ACKOR

3.10.1 The Behavioural Specification

`ACKOR` combines the acknowledgements for a single input port into a single bit. If any output acknowledges the input port, then an acknowledgement signal results.

```
⊢ ∀ ackTerm ackOut.
     ACKOR_SPEC (ackTerm,ackOut) = (∀ t. ackOut t = EXISTSABIT I (ackTerm t))
```

3.10.2 The Structural Specification

`ACKOR` corresponds almost directly to the HDL of the same name. `XiOR` is used as an OR gate with a word input rather than 4 single bit inputs.

```
⊢ ∀ ackTerm ackOut. ACKOR (ackTerm,ackOut) = XiOR (ackTerm,ackOut)
```

Qudos HDL

```
DEF ACKOR (ackTerm[0..3]: IN; ackOut: IO);

BEGIN
    OrAcks := XiOR4(ackTerm[0..3], ackOut);
END;
```

3.10.3 The Correctness Statement

The correctness statement is trivial to prove. It uses a basic fact about words that folding disjunction over a word (the definition of an OR gate) is equivalent to there existing a true bit in the word (the specification).

```
⊢ ∀ ackOut.
     FOR ackTerm ::(PSIGLEN (SUC n)).
       ACKOR (ackTerm,ackOut) ⊃ ACKOR_SPEC (ackTerm,ackOut)
```

3.11 ACKGEN

3.11.1 The Behavioural Specification

`ACKGEN` generates the acknowledgement signals for a single output port. If outputs are disabled for this port or the external port is sending a negative acknowledgement, then all inputs are sent a negative acknowledgement. Otherwise the granted input is sent an acknowledgement and the others negative acknowledgements.

```
⊢ ∀ n ackIn grant disabled.
     AckGen n ackIn grant disabled =
     ((disabled ∨ ~ ackIn) ⇒ (ZEROW n) |  (WPVEC n grant))
```

```
⊢ ∀ ackIn grant disabled ackTerm.
     ACKGEN_SPEC ((ackIn,grant,disabled),ackTerm) =
     (∀ t. ackTerm t = AckGen (SIGLEN ackTerm) (ackIn t) (grant t) (disabled t))
```

3.11.2 The Structural Specification

`ACKGEN` corresponds to the HDL of the same name. The x and y signals are combined as a single 2-bit word `grant`. Additional wires are used to name the two bits x and y. Otherwise the definitions are effectively the same.

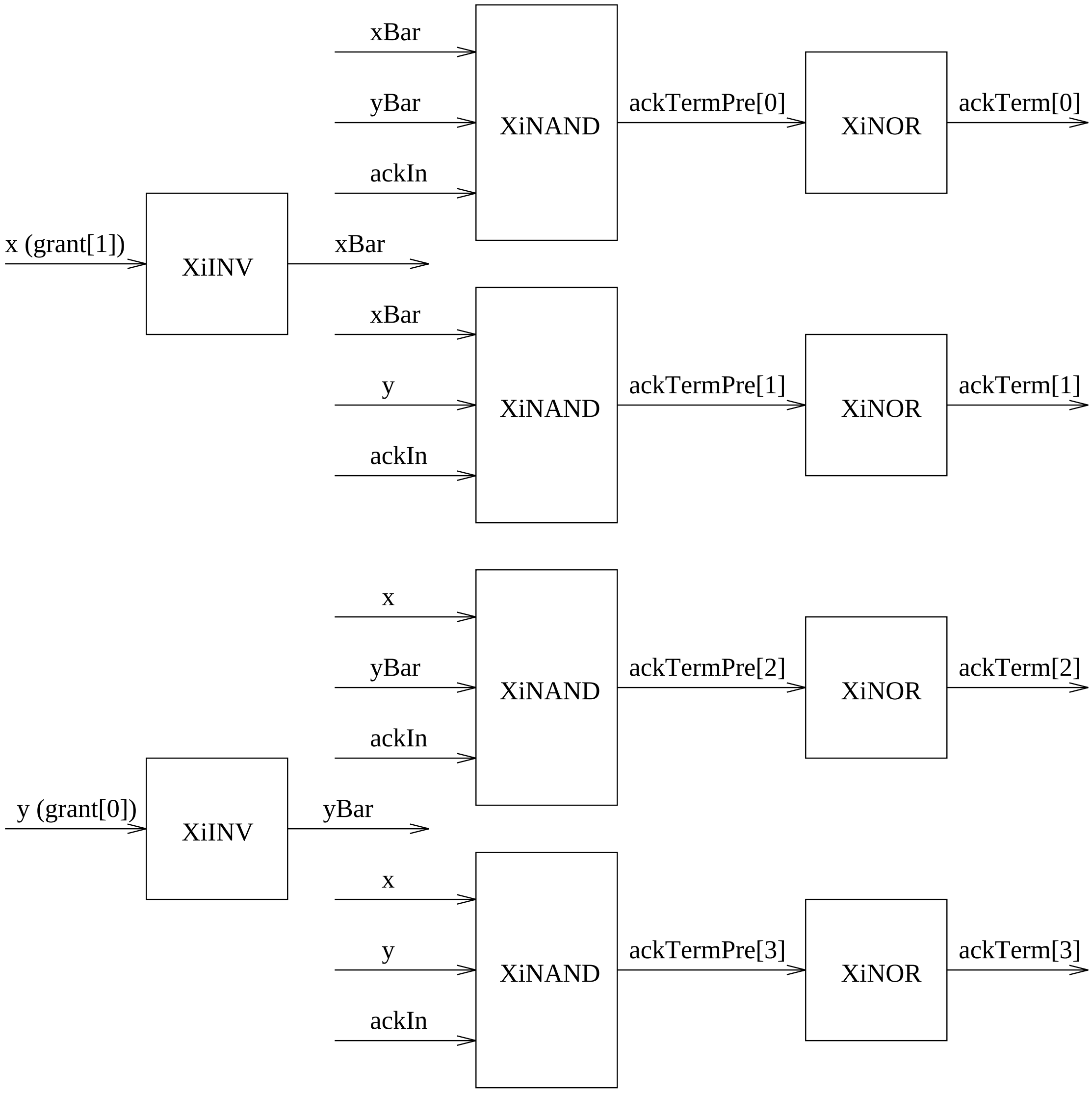

Figure 1: The Implementation of `ACKGEN`

```
⊢ ∀ ackIn grant disabled ackTerm.
     ACKGEN ((ackIn,grant,disabled),ackTerm) =
     (LOCAL x y xBar yBar.
       LOCAL ackTermPre ::(PSIGLEN 4).
         WIRE (SBIT 1 grant,x) ∧
         WIRE (SBIT 0 grant,y) ∧
         XiINV (x,xBar) ∧
         XiINV (y,yBar) ∧
         XiNAND (SWORD [xBar; yBar; ackIn],SBIT 0 ackTermPre) ∧
         XiNAND (SWORD [xBar; y; ackIn],SBIT 1 ackTermPre) ∧
         XiNAND (SWORD [x; yBar; ackIn],SBIT 2 ackTermPre) ∧
         XiNAND (SWORD [x; y; ackIn],SBIT 3 ackTermPre) ∧
         (FOR i ::(TO (SIGLEN ackTerm)).
           XiNOR (SWORD [SBIT i ackTermPre; disabled],SBIT i ackTerm)))
```

Qudos HDL

```
DEF ACKGEN (ackIn, x, y, disabled: IN; ackTerm[0..3] : IO);

xBar, yBar, ackTermPre[0..3]: IO;

BEGIN

   InvX := XiINV (x, xBar);
   InvY := XiINV (y, yBar);

   NandAckTermPre[0] := XiNAND3 (xBar, yBar, ackIn, ackTermPre[0]);
   NandAckTermPre[1] := XiNAND3 (xBar, y, ackIn, ackTermPre[1]);
   NandAckTermPre[2] := XiNAND3 (x, yBar, ackIn, ackTermPre[2]);
   NandAckTermPre[3] := XiNAND3 (x, y, ackIn, ackTermPre[3]);

   NorDis[0-3] := XiNOR2 (ackTermPre[0-3], disabled, ackTerm[0-3]);
END;
```

3.11.3 The Correctness Statement

Because the structural definition was not generic, the correctness theorem cannot be generic. The size of `grant` is thus fixed to be 2, and that of `ackTerm` is fixed to be 4 in the correctness statement.

```
⊢ ∀ ackIn disabled.
     FOR grant ::(PSIGLEN 2).
       FOR ackTerm ::(PSIGLEN 4).
         ACKGEN ((ackIn,grant,disabled),ackTerm) ⊃
         ACKGEN_SPEC ((ackIn,grant,disabled),ackTerm)
```

3.12 ACKOR_N

3.12.1 The Behavioural Specification

`ACKOR_N` combines the acknowledgements for all the input ports giving one bit per input port. If any output acknowledges an input, then an acknowledgement signal is sent to that input.

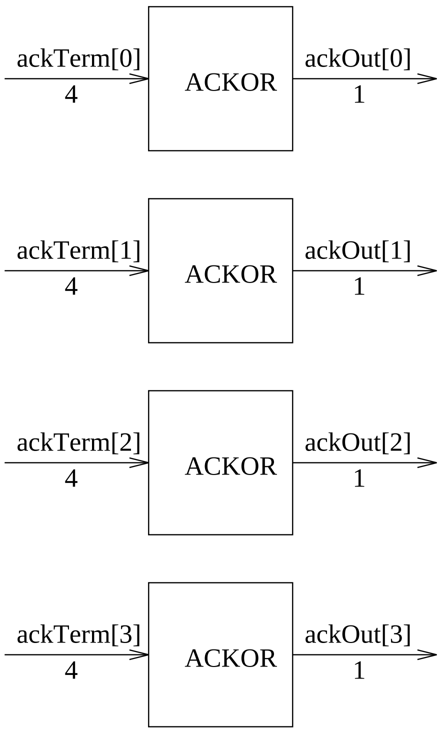

Figure 2: The Implementation of `ACKOR_N`

```
⊢ ∀ ackTerm ackOut.
     ACKOR_N_SPEC (ackTerm,ackOut) =
     (∀ t. ackOut t = WMAP (EXISTSABIT I) (ackTerm t))
```

3.12.2 The Structural Specification

`ACKOR_N` gives an extra level of hierarchy over the HDL version. It describes the 4 different occurrences of `ACKOR`. The input `ackTerm` is a word of 4 4-bit words. Each occurrence takes one of these words as argument. Due to the extra flexibility of HOL a loop construct can be used to describe this. This makes the structural description generic: the number of occurrences of `ACKOR` does not need to be fixed. It must just be the size of the argument word.

```
⊢ ∀ ackTerm ackOut.
     ACKOR_N (ackTerm,ackOut) =
     (FOR i ::(TO (SIGLEN ackOut)). ACKOR (SBIT i ackTerm,SBIT i ackOut))
```

```
⊢ ∀ ackTerm ackOut.
     ACKOR_N_SIMPL (ackTerm,ackOut) =
     (FOR i ::(TO (SIGLEN ackOut)). ACKOR_SPEC (SBIT i ackTerm,SBIT i ackOut))
```

Qudos HDL

```
AckOr[0] := ACKOR (ackTerm[0..3], ackOut[0]);
AckOr[1] := ACKOR (ackTerm[4..7], ackOut[1]);
AckOr[2] := ACKOR (ackTerm[8..11], ackOut[2]);
AckOr[3] := ACKOR (ackTerm[12..15], ackOut[3]);
```

3.12.3 The Correctness Statement

As the structural description is generic with respect to word sizes, the correctness theorem can also be generic. Two restrictions are placed on the sizes. First, `ackOut` must be the same size as the top level of `ackTerm`—`n`. Second, the number of bits in each word of `ackTerm` must not be zero. This is specified by giving the size as `SUC m`—the successor of `m`. This is sufficient because a successor cannot be zero.

```
⊢ ∀ n m.
     FOR ackTerm ::(PSIG2LEN n (SUC m)).
       FOR ackOut ::(PSIGLEN n).
         ACKOR_N (ackTerm,ackOut) ⊃ ACKOR_N_SPEC (ackTerm,ackOut)
```

3.13 ACKGEN_N

3.13.1 The Behavioural Specification

`ACKGEN_N` generates the acknowledgement signals for the output ports. If outputs are disabled for a port or the corresponding external port is sending a negative acknowledgement, then all inputs are sent a negative acknowledgement from that port. Otherwise the granted input is sent an acknowledgement and the others negative acknowledgements from that port. The separate signals for an input port are combined to give a single acknowledgement by a separate module—`ACKOR_N`.

```
⊢ ∀ ackIn grant outputDisable ackTerm.
     ACKGEN_N_SPEC ((ackIn,grant,outputDisable),ackTerm) =
     (∀ t.
       ackTerm t =
       MKW2 (SIGLEN ackTerm) (BSIGLEN ackTerm)
         (λ i.
           AckGen (SIGLEN ackTerm) (SBIT i ackIn t) (SBIT i grant t)
             (SBIT i outputDisable t)))
```

3.13.2 The Structural Specification

`ACKGEN_N` gives an extra level of hierarchy over the HDL version. It describes the 4 different occurrences of `ACKOR`. The input `ackIn` is a 4-bit word. Each occurrence uses one bit. The `xGrant` and `yGrant` inputs are combined into a single input `grant` which is a word of 4 2-bit words. The output `ackTerm` consists of 4 4-bit words. The sizes are not explicitly fixed in the HOL definition of `ACKGEN_N`. However, the definition of `ACKGEN` does fix the size of its arguments, so this implicitly fixes the sizes of the inner words. For example, `grant` is fixed to be an n by 2 word for some n.

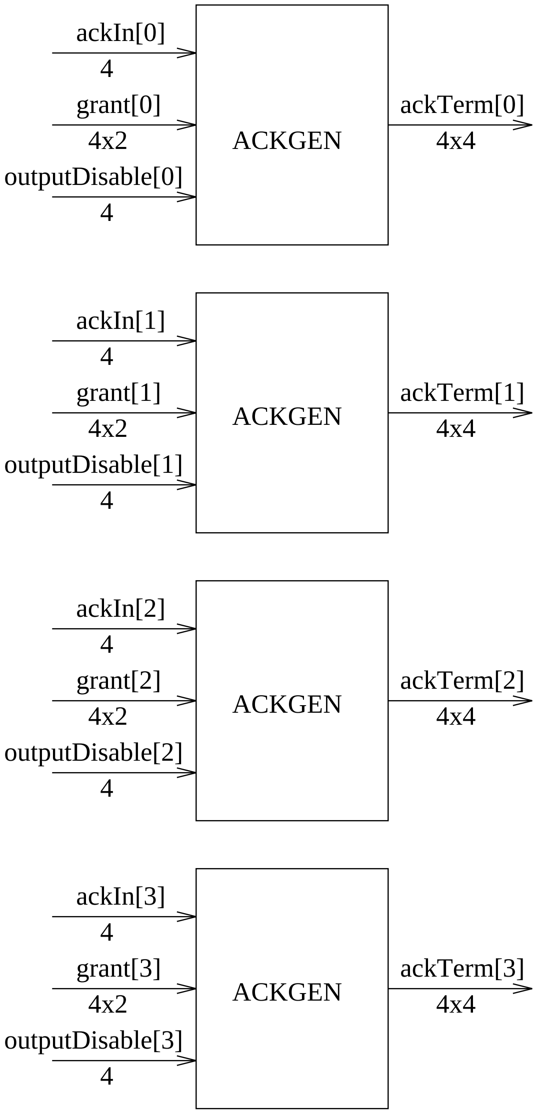

Figure 3: The Implementation of `ACKGEN_N`

```
⊢ ∀ ackIn grant outputDisable ackTerm.
     ACKGEN_N ((ackIn,grant,outputDisable),ackTerm) =
     (FOR i ::(TO (SIGLEN ackTerm)).
       ACKGEN
         ((SBIT i ackIn,SBIT i grant,SBIT i outputDisable),SBITS i ackTerm))
```

```
⊢ ∀ ackIn grant outputDisable ackTerm.
     ACKGEN_N_SIMPL ((ackIn,grant,outputDisable),ackTerm) =
     (FOR i ::(TO (SIGLEN ackTerm)).
       ACKGEN_SPEC
         ((SBIT i ackIn,SBIT i grant,SBIT i outputDisable),SBITS i ackTerm))
```

Qudos HDL

```
AckGen[0-3] := ACKGEN (ackIn[0-3], xGrant[0-3], yGrant[0-3],
                  outputDisable[0-3],
                  ackTerm[0-3], ackTerm[4-7], ackTerm[8-11], ackTerm[12-15]);
```

3.13.3 The Correctness Statement

The correctness theorem proved is specifically for a 4 by 4 switching element as the word sizes are fixed. It is assumed there are four input acknowledgement lines, four output disable lines, four grant signals each of two bits, specifying one of four input ports, and the output consists of four lines of four bits each.

```
⊢ FOR ackIn outputDisable ::(PSIGLEN 4).
     FOR grant ::(PSIG2LEN 4 2).
       FOR ackTerm ::(PSIG2LEN 4 4).
         ACKGEN_N ((ackIn,grant,outputDisable),ackTerm) ⊃
         ACKGEN_N_SPEC ((ackIn,grant,outputDisable),ackTerm)
```

3.14 `ACK`

3.14.1 The Behavioural Specification

`ACK` is the top-level specification of the Acknowledgement unit of the fabric. It produces the acknowledgement signals from the results of arbitration. It takes as input a word holding the acknowledgement signals from the output ports `ackIn`, a word indicating which request was granted for each output port `grant`, and a word of disable signals `outputDisable`, with one bit for each output port. It outputs a word of acknowledgement signals, `ackOut`, with one bit for each input port.

The bit of `ackOut` corresponding to each input port is specified separately using `Ackn`. It gives the acknowledgement signal for a single input port on a single clock cycle. The input port receives a positive acknowledgement provided it has been selected by some output port, `i`; the disable signal is not asserted indicating that the grant is valid at that time and provided the acknowledgement signal from the granted output port is positive. Otherwise a negative acknowledgement is seen.

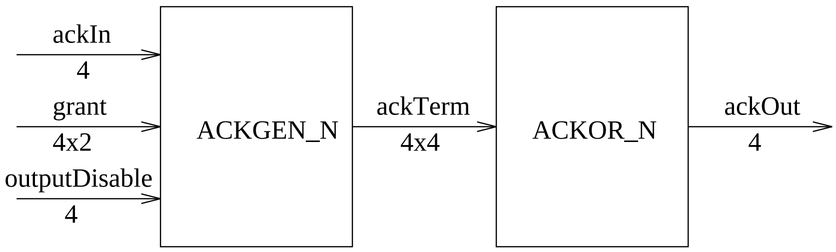

Figure 4: The Implementation of `ACK`

```
⊢ ∀ num_of_outs inportno ackIn grant disabled.
     Ackn num_of_outs inportno ackIn grant disabled =
     (LOCAL i.
       i < num_of_outs ∧
       ~ (BIT i disabled) ∧
       BIT i ackIn ∧
       (BNVAL (BIT i grant) = inportno))
```

```
⊢ ∀ ackIn grant outputDisable ackOut.
     ACK_SPEC ((ackIn,grant,outputDisable),ackOut) =
     (∀ t.
       ackOut t =
       MKW (SIGLEN ackOut)
         (λ i. Ackn (SIGLEN ackOut) i (ackIn t) (grant t) (outputDisable t)))
```

3.14.2 The Structural Specification

`ACK` gives an extra level of hierarchy over the HDL version. It combines all the acknowledgement hardware. The word sizes are fixed to be those of the 4x4 switching element.

```
⊢ ∀ ackIn grant outputDisable ackOut.
     ACK ((ackIn,grant,outputDisable),ackOut) =
     (LOCAL ackTerm ::(PSIG2LEN 4 4).
       ACKGEN_N ((ackIn,grant,outputDisable),ackTerm) ∧
       ACKOR_N (ackTerm,ackOut))
```

```
⊢ ∀ ackIn grant outputDisable ackOut.
     ACK_SIMPL ((ackIn,grant,outputDisable),ackOut) =
     (LOCAL ackTerm ::(PSIG2LEN 4 4).
       ACKGEN_N_SPEC ((ackIn,grant,outputDisable),ackTerm) ∧
       ACKOR_N_SPEC (ackTerm,ackOut))
```

Qudos HDL

```
AckGen[0-3] := ACKGEN (ackIn[0-3], xGrant[0-3], yGrant[0-3],
```

```
                  outputDisable[0-3],
                  ackTerm[0-3], ackTerm[4-7], ackTerm[8-11], ackTerm[12-15]);

AckOr[0] := ACKOR (ackTerm[0..3], ackOut[0]);
AckOr[1] := ACKOR (ackTerm[4..7], ackOut[1]);
AckOr[2] := ACKOR (ackTerm[8..11], ackOut[2]);
AckOr[3] := ACKOR (ackTerm[12..15], ackOut[3]);
```

3.14.3 The Correctness Statement

```
⊢ FOR ackIn ackOut outputDisable ::(PSIGLEN 4).
     FOR grant ::(PSIG2LEN 4 2).
       ACK ((ackIn,grant,outputDisable),ackOut) ⊃
       ACK_SPEC ((ackIn,grant,outputDisable),ackOut)
```

3.15 DMUX4T2FFC

3.15.1 The Behavioural Specification

`DMUX4T2FFC` selects 2 bits from a 4-bit word `d`. If `outputDisable` is set then a zero word is output. Otherwise, if signal `y` is false, it returns the 0^{th} and 2^{nd} bits. If signal `y` is true, it returns the 1^{st} and 3^{rd} bits. There is a single cycle delay.

```
⊢ ∀ d y outputDisable q.
     DMUX4T2FFC_SPEC ((d,y,outputDisable),q) =
     (∀ t.
       q (t + 1) =
       ((outputDisable t)
        ⇒ (ZEROW 2)
        |  (WORD [BIT (BV (y t) + 2) (d t); BIT (BV (y t)) (d t)])))
```

3.15.2 The Structural Specification

`DMUX4T2FFC` corresponds directly to the HDL version. The CLB statement is omitted since it is just a layout command. `RLATCH` is used for the flip-flops.

```
⊢ ∀ d y outputDisable q.
     DMUX4T2FFC ((d,y,outputDisable),q) =
     (LOCAL yBar.
       LOCAL mux ::(PSIGLEN 2).
         XiINV (y,yBar) ∧
         AO ((SBIT 0 d,yBar,SBIT 1 d,y),SBIT 0 mux) ∧
         AO ((SBIT 2 d,yBar,SBIT 3 d,y),SBIT 1 mux) ∧
         RLATCH ((mux,outputDisable),q))
```

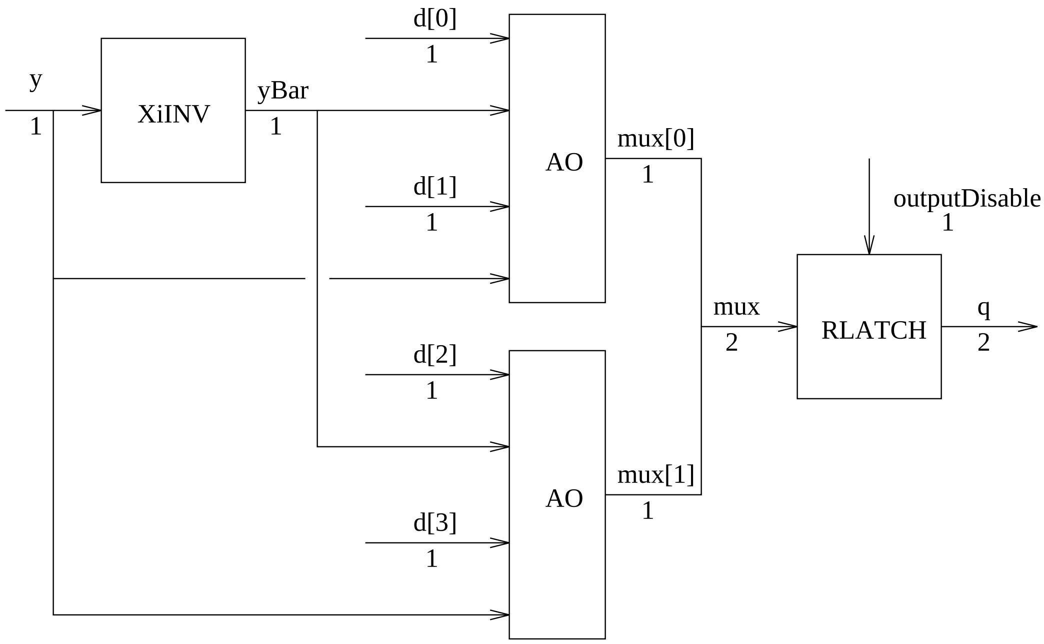

Figure 5: The Implementation of DMUX4T2FFC

```
⊢ ∀ d y outputDisable q.
     DMUX4T2FFC_SIMPL ((d,y,outputDisable),q) =
     (LOCAL yBar.
       LOCAL mux ::(PSIGLEN 2).
         XiINV (y,yBar) ∧
         AO ((SBIT 0 d,yBar,SBIT 1 d,y),SBIT 0 mux) ∧
         AO ((SBIT 2 d,yBar,SBIT 3 d,y),SBIT 1 mux) ∧
         RLATCH_SPEC ((mux,outputDisable),q))
```

Qudos HDL

```
DEF DMUX4T2FFC (d[0..3], clock, y, outputDisable: IN; q[0..1]: IO);
yBar, mux[0..1] : IO;

BEGIN
    Clb := XiCLBMAP5i2okr (d[0..1], y, d[2..3], clock,
     outputDisable, q[0..1]);
    InvY := XiINV(y, yBar);

    B[0] := AO (d[0], yBar, d[1], y, mux[0]);
    B[1] := AO (d[2], yBar, d[3], y, mux[1]);
    BFF[0-1] := XiDFFrd (mux[0-1], clock,
outputDisable, q[0-1]);
END;
```

3.15.3 The Correctness Statement

```
⊢ FOR q ::(PSIGLEN 2).
     DMUX4T2FFC ((d,y,outputDisable),q) ⊃
     DMUX4T2FFC_SPEC ((d,y,outputDisable),q)
```

3.16 DMUX4T2

3.16.1 The Behavioural Specification

DMUX4T2 is a 1-bit multiplexor, selecting either bit 0 or bit 1 of each of the words within a structured word. Which is selected depends on the value of the boolean signal x. It does not have a disable mode. It is not clocked and so causes no delay at this level of timing abstraction. It is defined using an auxiliary function Mux.

```
⊢ ∀ x d. Mux x d = BITS (BV x) d
```

```
⊢ ∀ d x dOut. DMUX4T2_SPEC ((d,x),dOut) = (∀ t. dOut t = Mux (x t) (d t))
```

3.16.2 The Structural Specification

DMUX4T2 corresponds to the HDL version. The CLB statement is omitted. The argument d is now a structured word holding 2 words of 2 bits each. This allows the two occurrences of AO to be combined using loop construct.

```
⊢ ∀ d x dOut.
     DMUX4T2 ((d,x),dOut) =
     (LOCAL xBar.
       XiINV (x,xBar) ∧
       (FOR i ::(TO 2).
         AO ((SBIT 0 (SBIT i d),xBar,SBIT 1 (SBIT i d),x),SBIT i dOut)))
```

Qudos HDL

```
DEF DMUX4T2 (d[0..3], x: IN; dOut[0..1]: IO);
xBar : IO;

BEGIN
    Clb := XiCLBMAP5i2o (d[0..1], x, d[2..3], dOut[0..1]);

    InvX := XiINV(x, xBar);

    B[0] := AO (d[0], xBar, d[1], x, dOut[0]);
    B[1] := AO (d[2], xBar, d[3], x, dOut[1]);
END;
```

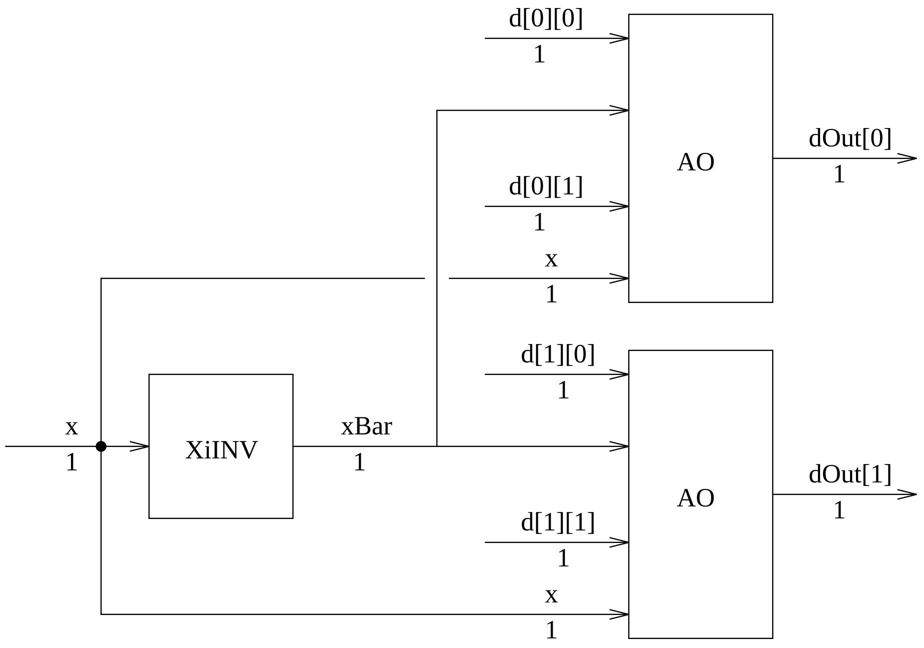

Figure 6: The Implementation of `DMUX4T2`

3.16.3 The Correctness Statement

```
⊢ ∀ x.
     FOR d ::(PSIG2LEN 2 2).
       FOR dOut ::(PSIGLEN 2).
         DMUX4T2 ((d,x),dOut) ⊃ DMUX4T2_SPEC ((d,x),dOut)
```

3.17 `DMUX2B4CAll`

3.17.1 The Behavioural Specification

`DMUX2B4CAll` chooses the `grant`th bit of each of the 2 words of which `d` consists. The two bits making up grant are sampled at different times. If `outputDisable` is set then a zero word is output. There is a single cycle delay.

It is defined in terms of the auxiliary function `DisableMux2`. If a disable signal `outputDisable` is set then the given `default` word is returned. Otherwise the result is a word composed of one bit from each of the words. The bit chosen is the same for each word and is given by the `select` signal. The select signal is converted to a number using the given conversion function, `to_num` (here `BNVAL`).

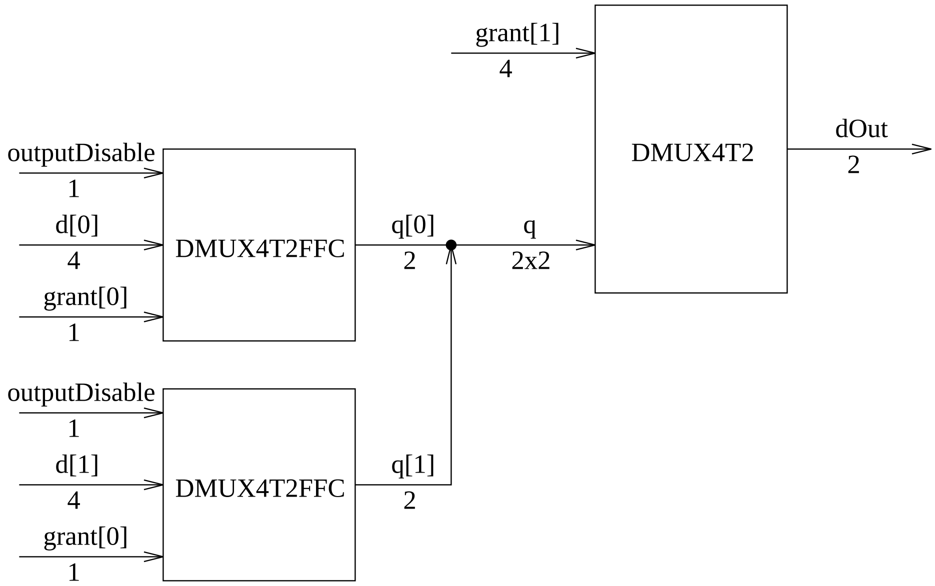

Figure 7: The Implementation of DMUX2B4CAll

```
⊢ ∀ to_num default d select outputDisable.
    DisableMux2 to_num default d select outputDisable =
    (outputDisable ⇒ default | (BITS (to_num select) d))
```

```
⊢ ∀ d grant outputDisable dOut.
    DMUX2B4CAll_SPEC ((d,grant,outputDisable),dOut) =
    (∀ t.
      dOut (t + 1) =
      DisableMux2 BNVAL (ZEROW 2) (d t)
        (WORD [BIT 1 (grant (t + 1)); BIT 0 (grant t)])
        (outputDisable t))
```

3.17.2 The Structural Specification

`DMUX2B4CAll` corresponds to the HDL version. The signals `d0` and `d1` have been combined into a word of 2 4-bit words, `d`. This allows the two occurrences of `DMUX4T2FFC` to be combined. Similarly, `x` and `y` have been combined into a 2-bit word, `grant`. This makes the semantics cleaner as it can be given in terms of the number represented by `grant`. Finally, the signals `q0` and `q1` have been combined into a word of 2 2-bit words, `q`.

```
⊢ ∀ d grant outputDisable dOut.
     DMUX2B4CAll ((d,grant,outputDisable),dOut) =
     (LOCAL x y.
       LOCAL q ::(PSIG2LEN 2 2).
         WIRE (SBIT 1 grant,x) ∧
         WIRE (SBIT 0 grant,y) ∧
         (FOR i ::(TO 2). DMUX4T2FFC ((SBIT i d,y,outputDisable),SBIT i q)) ∧
         DMUX4T2 ((q,x),dOut))
```

```
⊢ ∀ d grant outputDisable dOut.
     DMUX2B4CAll_SIMPL ((d,grant,outputDisable),dOut) =
     (LOCAL x y.
       LOCAL q ::(PSIG2LEN 2 2).
         WIRE (SBIT 1 grant,x) ∧
         WIRE (SBIT 0 grant,y) ∧
         (FOR i ::(TO 2).
           DMUX4T2FFC_SPEC ((SBIT i d,y,outputDisable),SBIT i q)) ∧
         DMUX4T2_SPEC ((q,x),dOut))
```

Qudos HDL

```
DEF DMUX2B4CAll (d0[0..3], d1[0..3], clock, x, y, outputDisable: IN;
                 dOut[0..1]: IO);

q0[0..1], q1[0..1]: IO;

BEGIN
    P[0] := DMUX4T2FFC (d0[0..3], clock, y, outputDisable, q0[0..1]);
    P[1] := DMUX4T2FFC (d1[0..3], clock, y, outputDisable, q1[0..1]);
    FB := DMUX4T2 (q0[0..1], q1[0..1], x, dOut[0..1]);
END;
```

3.17.3 The Correctness Statement

```
⊢ ∀ outputDisable.
     FOR d ::(PSIG2LEN 2 4).
       FOR q y ::(PSIGLEN 2).
         DMUX2B4CAll ((d,y,outputDisable),q) ⊃
         DMUX2B4CAll_SPEC ((d,y,outputDisable),q)
```

3.18 DATASWITCHC

3.18.1 The Behavioural Specification

`DATASWITCHC` chooses the `grant`th word from the 4 words of which `d` consists. The two bits of which grant consists are sampled at different times. If `outputDisable` is set then a zero word is output. There is a single cycle delay.

It is defined in terms of `DisableMux`. If a disable signal `outputDisable` is set then the given `default` word is returned. Otherwise the word or signal in the bit position of data `d` indicated by the `select` signal is returned. The select signal is converted to a number using the given conversion function `to_num`, here `BNVAL`.

```
⊢ ∀ to_num default d select outputDisable.
     DisableMux to_num default d select outputDisable =
     (outputDisable ⇒ default |  (BIT (to_num select) d))
```

```
⊢ ∀ d grant outputDisable q.
     DATASWITCHC_SPEC ((d,grant,outputDisable),q) =
     (∀ t.
       q (t + 1) =
       DisableMux BNVAL (ZEROW (SIGLEN q)) (d t)
         (WORD [BIT 1 (grant (t + 1)); BIT 0 (grant t)])
         (outputDisable t))
```

3.18.2 The Structural Specification

`DATASWITCHC` differs in several ways from the HDL. The data input is structured into 8-bit bytes instead of 32 bits. Thus the bit positions are the same modulo 8. For example, the HDL `(d[0], d[8], d[16], d[24],...)` becomes `SBITS 0 d` (though see below), since each of those bits is the zeroth bit of its byte. The individual bits are combined into a single 4-bit word. The x and y signals are passed as a single 2-bit word `grant`.

The 4 occurrences of `d` are combined into a single parameterised call. For example, the HDL `(d[0], d[8], d[16], d[24],...)` actually becomes `SBITS (2 * i) d` where `i` is 0 and similarly for `(d[2], d[10], d[18], d[26],...)` where `i` is 1. These are combined into a single word of 2 4-bit words. Note the order they are written is reversed. This is because the HOL word library numbers words from the right hand side.

```
⊢ ∀ d grant outputDisable q.
     DATASWITCHC ((d,grant,outputDisable),q) =
     (FOR i ::(TO (SIGLEN d)).
       DMUX2B4CAll
         ((SWORD [SBITS (2 * i + 1) d; SBITS (2 * i) d],grant,outputDisable),
          SWSEG 2 (2 * i) q))
```

```
⊢ ∀ d grant outputDisable q.
     DATASWITCHC_SIMPL ((d,grant,outputDisable),q) =
     (FOR i ::(TO (SIGLEN d)).
       DMUX2B4CAll_SPEC
         ((SWORD [SBITS (2 * i + 1) d; SBITS (2 * i) d],grant,outputDisable),
          SWSEG 2 (2 * i) q))
```

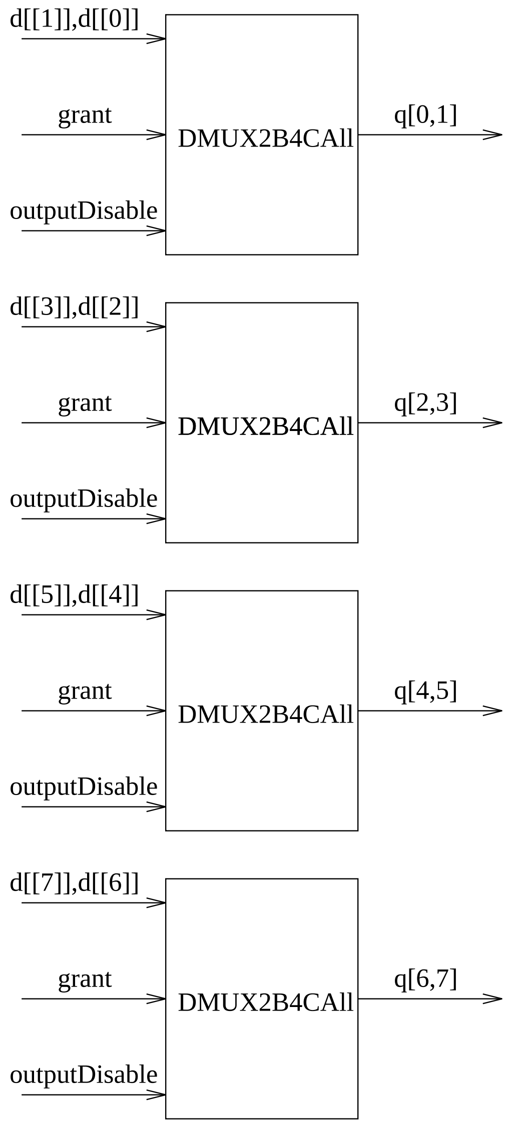

Figure 8: The Implementation of `DATASWITCHC`

Qudos HDL

```
(* Version that allows all bits to be cleared  *)
DEF DATASWITCHC (d[0..31], clock, x, y, outputDisable: IN; q[0..7]: IO);
BEGIN
    Pr[0] := DMUX2B4CAll (d[0], d[8], d[16], d[24],
                          d[1], d[9], d[17], d[25],
                          clock, x, y, outputDisable, q[0], q[1]);

    Pr[1] := DMUX2B4CAll (d[2], d[10], d[18], d[26],
                        d[3], d[11], d[19], d[27],
                        clock, x, y, outputDisable, q[2], q[3]);

    Pr[2] := DMUX2B4CAll (d[4], d[12], d[20], d[28],
                        d[5], d[13], d[21], d[29],
                        clock, x, y, outputDisable, q[4], q[5]);

    Pr[3] := DMUX2B4CAll (d[6], d[14], d[22], d[30],
                        d[7], d[15], d[23], d[31],
                        clock, x, y, outputDisable, q[6], q[7]);
END;
```

3.18.3 The Correctness Statement

```
⊢ FOR d ::(PSIG2LEN 4 8).
     FOR q ::(PSIGLEN 8).
       FOR grant ::(PSIGLEN 2).
         DATASWITCHC ((d,grant,outputDisable),q) ⊃
         DATASWITCHC_SPEC ((d,grant,outputDisable),q)
```

3.19 DATASWITCH_N

3.19.1 The Behavioural Specification

`DATASWITCH_N` chooses a word to be output to each of the output ports. For each output there is a separate grant and disable word within `grant` and `outputDisable`, respectively. For each output the `grant`th word from `d` is selected. If `outputDisable` is set then a zero word is output. There is a single cycle delay.

```
⊢ ∀ d grant outputDisable dOut.
     DATASWITCH_N_SPEC ((d,grant,outputDisable),dOut) =
     (∀ t.
       dOut (t + 1) =
       MKW (SIGLEN dOut)
         (λ i.
           DisableMux BNVAL (ZEROW (BSIGLEN dOut)) (d t)
             (WORD [BIT 1 (BIT i (grant (t + 1))); BIT 0 (BIT i (grant t))])
             (BIT i (outputDisable t))))
```

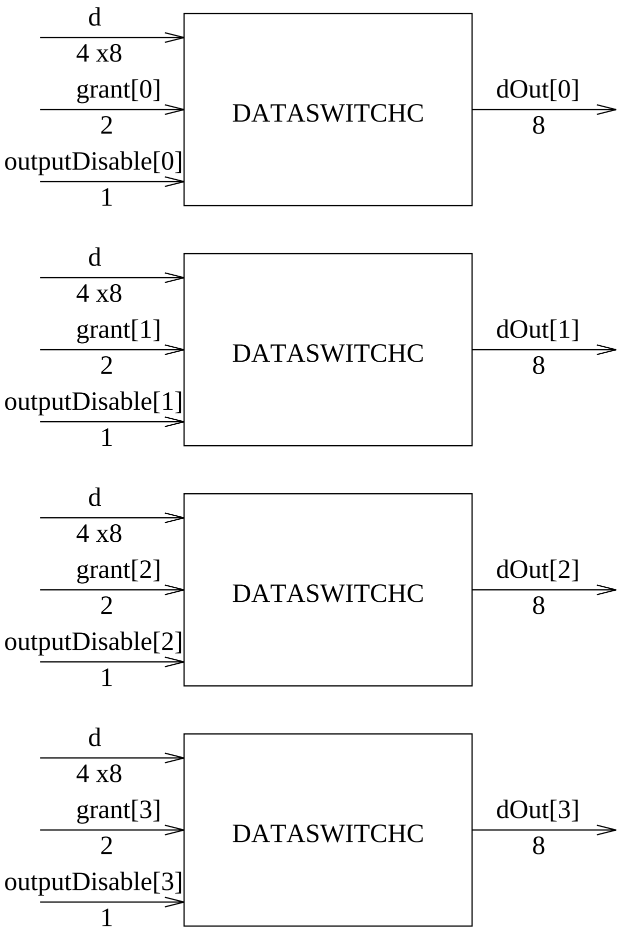

Figure 9: The Implementation of `DATASWITCH_N`

3.19.2 The Structural Specification

`DATASWITCH_N` gives an extra level of hierarchy over the HDL version. It describes the 4 different occurrences of the `DATASWITCH`. The clock signal is omitted as it is abstracted away from in the semantics. The signals `xGrant` and `yGrant` are combined into a word of 4 2-bit words, `grant`. The `dPause` and `dOut` signals are grouped into words of bytes.

```
⊢ ∀ d grant outputDisable dOut.
     DATASWITCH_N ((d,grant,outputDisable),dOut) =
     (FOR i ::(TO (SIGLEN dOut)).
       DATASWITCHC ((d,SBIT i grant,SBIT i outputDisable),SBIT i dOut))
```

```
⊢ ∀ d grant outputDisable dOut.
     DATASWITCH_N_SIMPL ((d,grant,outputDisable),dOut) =
     (FOR i ::(TO (SIGLEN dOut)).
       DATASWITCHC_SPEC ((d,SBIT i grant,SBIT i outputDisable),SBIT i dOut))
```

Qudos HDL

```
DSw[0] := DATASWITCH (dPause[0..31], clock, xGrant[0], yGrant[0],
                                outputDisable[0], dOut[0..7]);
DSw[1] := DATASWITCH (dPause[0..31], clock, xGrant[1], yGrant[1],
                                outputDisable[1], dOut[8..15]);
DSw[2] := DATASWITCH (dPause[0..31], clock, xGrant[2], yGrant[2],
                                outputDisable[2], dOut[16..23]);
DSw[3] := DATASWITCH (dPause[0..31], clock, xGrant[3], yGrant[3],
                                outputDisable[3], dOut[24..31]);
```

3.19.3 The Correctness Statement

```
⊢ FOR grant ::(PSIG2LEN 4 2).
     FOR dOut d ::(PSIG2LEN 4 8).
       DATASWITCH_N ((d,grant,outputDisable),dOut) ⊃
       DATASWITCH_N_SPEC ((d,grant,outputDisable),dOut)
```

3.20 PAUSE_DATASWITCH

3.20.1 The Behavioural Specification

`PAUSE_DATASWITCH` chooses a word to be output to each of the output ports. It delays the data long enough for an arbitration decision to be made. For each output there is a separate grant and disable word within `grant` and `outputDisable`, respectively. For each output the `grant`th word from `d` is selected. If `outputDisable` is set then a zero word is output. There is a two cycle delay on the data line, though the control lines affect the output with only a single cycle delay.

```
⊢ ∀ d grant outputDisable dOut.
     PAUSE_DATASWITCH_SPEC ((d,grant,outputDisable),dOut) =
     (∀ t.
       dOut (t + 2) =
       MKW (SIGLEN dOut)
         (λ i.
           DisableMux BNVAL (ZEROW (BSIGLEN dOut)) (d t)
             (WORD
               [BIT 1 (BIT i (grant (t + 2))); BIT 0 (BIT i (grant (t + 1)))])
             (BIT i (outputDisable (t + 1)))))
```

3.20.2 The Structural Specification

`PAUSE_DATASWITCH` gives an extra level of hierarchy over the HDL version. It groups the Dataswitch and Pause register together into a single unit.

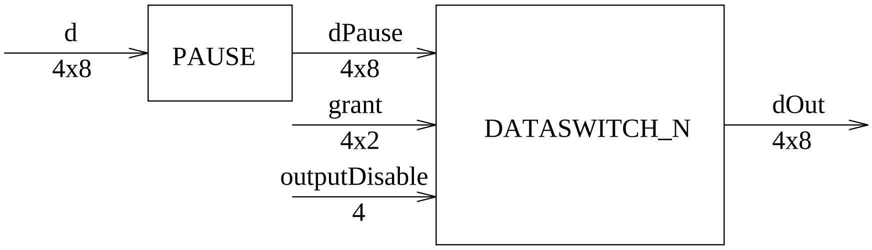

Figure 10: The Implementation of PAUSE_DATASWITCH

```
⊢ ∀ d grant outputDisable dOut.
     PAUSE_DATASWITCH ((d,grant,outputDisable),dOut) =
     (LOCAL dPause ::(PSIG2LEN 4 8).
       PAUSE (d,dPause) ∧ DATASWITCH_N ((dPause,grant,outputDisable),dOut))
```

```
⊢ ∀ d grant outputDisable dOut.
     PAUSE_DATASWITCH_SIMPL ((d,grant,outputDisable),dOut) =
     (LOCAL dPause ::(PSIG2LEN 4 8).
       PAUSE_SPEC (d,dPause) ∧
       DATASWITCH_N_SPEC ((dPause,grant,outputDisable),dOut))
```

Qudos HDL

```
Pause[0-31] := XiDFFd(d[0-31], clock, dPause[0-31]);

DSw[0] := DATASWITCH (dPause[0..31], clock, xGrant[0], yGrant[0],
                                outputDisable[0], dOut[0..7]);
DSw[1] := DATASWITCH (dPause[0..31], clock, xGrant[1], yGrant[1],
                                outputDisable[1], dOut[8..15]);
DSw[2] := DATASWITCH (dPause[0..31], clock, xGrant[2], yGrant[2],
                                outputDisable[2], dOut[16..23]);
DSw[3] := DATASWITCH (dPause[0..31], clock, xGrant[3], yGrant[3],
                                outputDisable[3], dOut[24..31]);
```

3.20.3 The Correctness Statement

```
⊢ FOR grant ::(PSIG2LEN 4 2).
     FOR dOut d ::(PSIG2LEN 4 8).
       PAUSE_DATASWITCH ((d,grant,outputDisable),dOut) ⊃
       PAUSE_DATASWITCH_SPEC ((d,grant,outputDisable),dOut)
```

3.21 TIMING

3.21.1 The Behavioural Specification

TIMING determines when the arbitration unit is triggered. The routeEnable signal is normally low.

After a frameStart signal it waits until any of the active bits go high, indicating the start of a packet. On the next cycle routeEnable goes high for one cycle, returning to low until the next frame. This is described by the abstract specification TIMING_SPEC. This specification is at the frame time level. It defines the output values over the period of a frame.

```
⊢ ∀ frameStart active routeEnable.
     TIMING_SPEC ((frameStart,active),routeEnable) =
     (∀ ts ta te.
       ~ (frameStart 0) ∧
       (∀ t. ~ (frameStart (t + 1)) ∨ ~ (EXISTSABIT I (active t))) ⊃
       (∀ t. ~ (frameStart t ∧ routeEnable t)) ∧
       (AFRAME2 ts ta te frameStart active ⊃
        STABLE (ts + 1) (ta + 1) routeEnable F ∧
        (routeEnable (ta + 1) = T) ∧
        STABLE (ta + 2) (te + 1) routeEnable F) ∧
       (IFRAME ts te frameStart active ⊃
        STABLE (ts + 1) (te + 1) routeEnable F))
```

The routeEnable signal must not occur at the same time as the frameStart signal if the arbiter is to function correctly. If it were to occur, TIMING would act as though it had received a cell in the previous frame, but the arbiter would act as though the cell was in the next frame. We include this condition in the specification. It can be ensured if an active bit does not arrive in the cycle prior to a frame start. An explicit assumption in the specification states that it does not occur.

```
∀t.  ~(frameStart (t+1)) ∨ ~(EXISTSABIT I (active t))
```

We also assume that the frame start does not occur at the initial time, as if so, the internal state on power up could be such that routeEnable and frameStart do occur together.

```
~(frameStart 0)
```

Furthermore, the frame start signal should not arrive at the same time as the active signal. If this occurs the active signal will just be ignored. This is because at the time ta+1 the route enable signal should go high for one cycle. However, at time ts+1 the frame start forces it low. Thus if ts=ta the active signal is ignored. This is reflected in the way an active frame is defined. AFRAME2 includes the clause ts < ta.

A more concrete specification, closer to the implementation is given by TIMING_SPEC2. This specification is at the clock cycle level. It gives a simple logical expression relating the values of the outputs on a cycle to the values on the inputs and outputs on the previous cycle. It describes the unit in terms of 2 bits of state xy. It is normally in a RUN state (xy = 00). On a frameStart signal it moves to a WAIT state (xy = 01). On an active signal it moves to a ROUTE state (xy = 11). One cycle later it returns to the RUN state. routeEnable corresponds to the x signal so is high only when in the ROUTE state. This specification allows the correctness proof to be split into two much simpler parts.

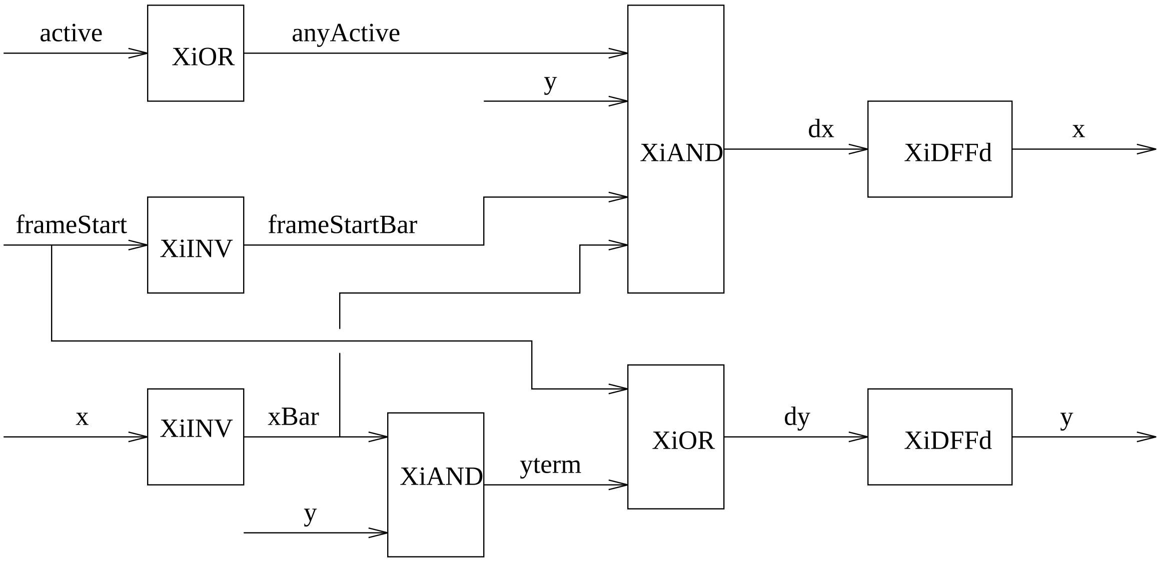

Figure 11: The Implementation of `TIMING`

```
⊢ ∀ y frameStart active x.
     TIMING_SPEC2 y ((frameStart,active),x) =
     (∀ t.
       (x (t + 1) =
        ~ (x t) ∧ y t ∧ EXISTSABIT I (active t) ∧ ~ (frameStart t)) ∧
       (y (t + 1) = ~ (x t) ∧ y t ∨ frameStart t))
```

3.21.2 The Structural Specification

`TIMING` corresponds almost directly to the HDL macro.

```
⊢ ∀ frameStart active x.
     TIMING ((frameStart,active),x) =
     (LOCAL xBar y dx dy yterm anyActive frameStartBar.
       XiOR (active,anyActive) ∧
       XiINV (frameStart,frameStartBar) ∧
       XiAND (SWORD [xBar; y],yterm) ∧
       XiOR (SWORD [yterm; frameStart],dy) ∧
       XiAND (SWORD [xBar; y; anyActive; frameStartBar],dx) ∧
       XiDFFd (dx,x) ∧
       XiINV (x,xBar) ∧
       XiDFFd (dy,y))
```

Qudos HDL

```
DEF TIMING (frameStart, clock, active [0..3]: IN;
            x (* = routeEnable *) : IO);
```

```
xBar, y, dx, dy, yterm,
anyActive, frameStartBar : IO;

BEGIN
    OrAnyActive := XiOR4 (active[0..3], anyActive);
    InvFS := XiINV (frameStart, frameStartBar);

    AndyTerm := XiAND2 (xBar, y, yterm);
    OrDy := XiOR2 (yterm, frameStart, dy);

    AndDx := XiAND4 (xBar, y, anyActive, frameStartBar, dx);


    FFx := XiDFFd (dx, clock, x);
    InvX := XiINV (x, xBar);

    FFy := XiDFFd (dy, clock, y);
END;
```

3.21.3 The Correctness Statement

The correctness of the timing module is proved in several parts. First, it is proved that the implementation implements the second specification given above for some value of the internal state y. This is trivial.

```
⊢ ∀ frameStart x.
     FOR active ::(PSIGLEN (SUC n)).
       TIMING ((frameStart,active),x) ⊃
       (∃ y. TIMING_SPEC2 y ((frameStart,active),x))
```

We then prove that the assumptions made in the first specification, together with the specification of the `routeEnable` signal from the second lower level specification (where it is named `x`) imply that the `frameStart` and `routeEnable` signals never occur together. This is again very simple.

```
⊢ ∀ frameStart routeEnable active.
     (∀ t.
       routeEnable (t + 1) =
       ~ (routeEnable t) ∧
       y t ∧
       EXISTSABIT I (active t) ∧
       ~ (frameStart t)) ∧
     ~ (frameStart 0) ∧
     (∀ t. ~ (frameStart (t + 1)) ∨ ~ (EXISTSABIT I (active t))) ⊃
     (∀ t. ~ (frameStart t ∧ routeEnable t))
```

We then prove that the second specification implies the first, for some value of the internal state.

```
⊢ ∀ frameStart x.
    FOR active ::(PSIGLEN (SUC n)).
      (∃ y. TIMING_SPEC2 y ((frameStart,active),x)) ⊃
      TIMING_SPEC ((frameStart,active),x)
```

The proof splits into several cases corresponding to the separate clauses in the high level specification. The first corresponds to the requirement that the `frameStart` and `routeEnable` signals never occur together and so follows from the above theorem. The remaining cases are concerned with the behaviour over a frame. One case corresponds to the behaviour over an inactive cycle. The remaining three correspond to active cycles: up to the active time, on the cycle just after the active time and from this point to the end of the frame. We give an informal overview of why these cases hold below.

In an inactive frame, or active frame up to the active time, the `routeEnable` signal is forced low because the active signal is low on the previous cycle throughout the period. At time `ta` the `routeEnable` signal is low, the active signal is high in at least one bit, and the `frameStart` signal is low. Therefore from the definition `TIMING_SPEC2`, for the `routeEnable` signal to be high at `ta+1`, we need `(y ta)` to be high. This will be so because it is set by the `frameStart` signal, and once it is high remains so whilst `routeEnable` is low. Thus the `routeEnable` signal will be high at time `ta+1`. Because `routeEnable` is high at time `ta+1`, it will be low at time `ta+2`. It will then remain low for the remainder of the cycle because `y` is low.

Finally the above lemmas are trivially combined to give the final correctness theorem that the structural specification implies the behavioural specification in terms of frames.

```
⊢ ∀ frameStart x.
    FOR active ::(PSIGLEN (SUC n)).
      TIMING ((frameStart,active),x) ⊃ TIMING_SPEC ((frameStart,active),x)
```

3.22 PRIFILUNIT

3.22.1 The Behavioural Specification

`PRIFILUNIT` performs priority filtering on the request of a single input port for a single output port. It takes as input two booleans, `hiR` and genR. The first indicates whether the input port in question is making a high priority request for this output port. The second indicates whether it is making a general request. The status of the requests from each of the other ports `hiOtherR` is also input. If there are any high priority requests for this output port, then the high priority value for this input port is output on the boolean signal `req`, otherwise the general request value is output.

```
⊢ ∀ hiR genR hiOtherR req.
    PRIFILUNIT_SPEC ((hiR,genR,hiOtherR),req) =
    (∀ t. req t = ((hiR t ∨ EXISTSABIT I (hiOtherR t)) ⇒ (hiR t) |  (genR t)))
```

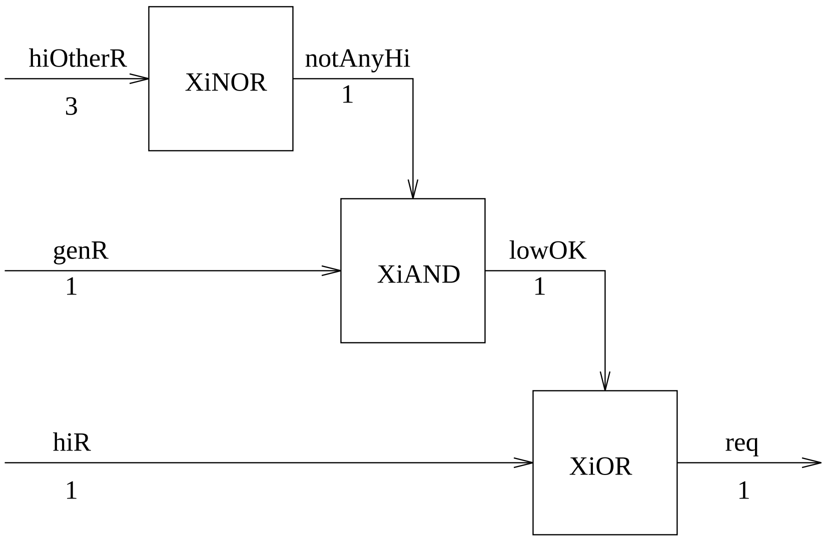

Figure 12: The Implementation of PRIFILUNIT

3.22.2 The Structural Specification

PRIFILUNIT corresponds directly to the HDL macro.

```
⊢ ∀ hiR genR hiOtherR req.
     PRIFILUNIT ((hiR,genR,hiOtherR),req) =
     (LOCAL notAnyHi lowOK.
       XiNOR (hiOtherR,notAnyHi) ∧
       XiAND (SWORD [notAnyHi; genR],lowOK) ∧
       XiOR (SWORD [lowOK; hiR],req))
```

Qudos HDL

```
DEF PRIFILUNIT (hiR, genR, hiOtherR[0..2]: IN; req: IO);

notAnyHi, lowOK: IO;

BEGIN
   NorAnyHi := XiNOR3 (hiOtherR[0..2], notAnyHi);
   AndLowOK := XiAND2 (notAnyHi, genR, lowOK);
   OrReq := XiOR2 (lowOK, hiR, req);
END;
```

3.22.3 The Correctness Statement

The correctness theorem is generic with respect to the size of word hiOtherR. It holds provided there is at least one other port competing for resources.

```
⊢ ∀ hiR genR req n.
    FOR hiOtherR ::(PSIGLEN (SUC n)).
      PRIFILUNIT ((hiR,genR,hiOtherR),req) ⊃
      PRIFILUNIT_SPEC ((hiR,genR,hiOtherR),req)
```

3.23 PRIFIL4CLB_SPEC

3.23.1 The Behavioural Specification

PRIFIL4CLB performs priority filtering for a single output port. It takes as input two words, hiR and genR. Each contains one bit for each input port. hiR indicates which input ports are making high priority requests for this output port. genR indicates which are making general requests. If there are any high priority requests for this output port, then the high priority requests are output to req, otherwise the general requests are output. Each bit position of req indicates the request from one input port.

PriFilt gives the function of a single slice (that is for one output port) of the priority filter for a single clock cycle.

```
⊢ ∀ hiR genR. PriFilt hiR genR = ((EXISTSABIT I hiR) ⇒ hiR | genR)
```

```
⊢ ∀ hiR genR req.
    PRIFIL4CLB_SPEC ((hiR,genR),req) = (∀ t. req t = PriFilt (hiR t) (genR t))
```

3.23.2 The Structural Specification

PRIFIL4CLB corresponds to the HDL macro of the same name. The 4 instances of PRIFILUNIT are replaced by a single higher-order call. To allow this, the function SREMBIT is used. It returns a word of size one smaller than its argument, with the given bit removed.

```
⊢ ∀ hiR genR req.
    PRIFIL4CLB ((hiR,genR),req) =
    (FOR i ::(TO (SIGLEN req)).
      PRIFILUNIT ((SBIT i hiR,SBIT i genR,SREMBIT i hiR),SBIT i req))
```

```
⊢ ∀ hiR genR req.
    PRIFIL4CLB_SIMPL ((hiR,genR),req) =
    (FOR i ::(TO (SIGLEN req)).
      PRIFILUNIT_SPEC ((SBIT i hiR,SBIT i genR,SREMBIT i hiR),SBIT i req))
```

Qudos HDL

```
DEF PRIFIL4CLB (hiR[0..3], genR[0..3]: IN; req[0..3]: IO);
BEGIN
```

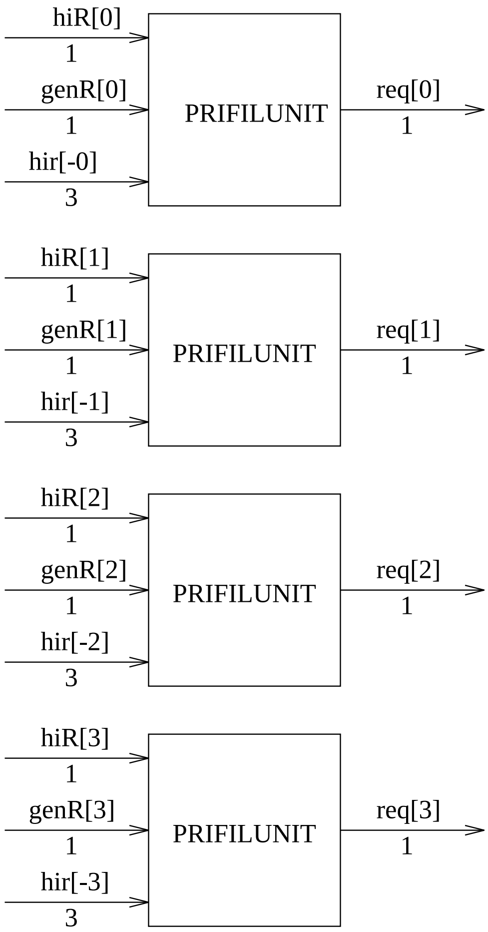

Figure 13: The Implementation of `PRIFIL4CLB`

```
   PriUnit[0] := PRIFILUNIT (hiR[0], genR[0], hiR[1..3], req[0]);
   PriUnit[1] := PRIFILUNIT (hiR[1], genR[1], hiR[0], hiR[2..3], req[1]);
   PriUnit[2] := PRIFILUNIT (hiR[2], genR[2], hiR[0..1], hiR[3], req[2]);
   PriUnit[3] := PRIFILUNIT (hiR[3], genR[3], hiR[0..2], req[3]);
END;
```

3.23.3 The Correctness Statement

The correctness theorem is proved in three stages. First we prove a generic theorem that the structural specification given in terms of the specification of its parts implements the behavioural specification. This is a generic theorem in which the word sizes of the inputs and outputs is not specified. All that is important is that they have the same non-zero size.

```
⊢ ∀ n.
     FOR hiR ::(PSIGLEN (SUC n)).
       FOR genR ::(PSIGLEN (SUC n)).
         FOR req ::(PSIGLEN (SUC n)).
           PRIFIL4CLB_SIMPL ((hiR,genR),req) ⊃
           PRIFIL4CLB_SPEC ((hiR,genR),req)
```

We then prove that the structural specification written in terms of the specification of its parts is implemented by that given in terms of the implementation of the parts. This uses the correctness statements of the parts which state that the implementations of the parts do implement their behavioural specifications, so can be used interchangeably. Since we are now dealing with a fixed implementation, the words lengths are rigidly defined to be 4.

```
⊢ FOR hiR genR req ::(PSIGLEN 4).
     PRIFIL4CLB ((hiR,genR),req) ⊃ PRIFIL4CLB_SPEC ((hiR,genR),req)
```

Finally, we trivially combine these two theorems to give the required correctness theorem, stating that the implementation does implement the specification.

```
⊢ FOR hiR req ::(PSIGLEN 4).
     PRIFIL4CLB ((hiR,genR),req) ⊃ PRIFIL4CLB_SIMPL ((hiR,genR),req)
```

3.24 PRIORITY

3.24.1 The Behavioural Specification

`PRIORITY` is the top-level specification of the Priority Filter block of the fabric. It takes as input two words of words, `hiR` and `genR`. Each contains one word for each input port, with that inner word containing one bit for each output port. `hiR` indicates which input ports are making high priority requests for which output ports. `genR` indicates which are making general requests.

It outputs a word of words, `req`. This contains one word for each input port. Each inner word has a position for each output port. It indicates which output port, if any, this input port is making a request for. The request must be of as high a priority as any other request for that output port.

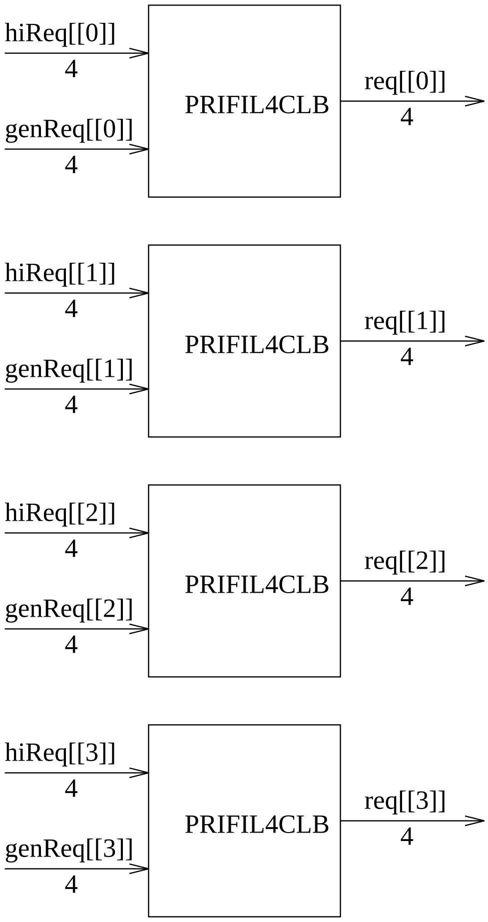

Figure 14: The Implementation of PRIORITY

```
⊢ ∀ hiReq genReq req.
     PRIORITY_SPEC ((hiReq,genReq),req) =
     (∀ t.
       req t =
       MKW2 (SIGLEN req) (BSIGLEN req)
         (λ n. PriFilt (BITS n (hiReq t)) (BITS n (genReq t))))
```

3.24.2 The Structural Specification

PRIORITY gives an extra level of hierarchy over the HDL version. It describes the 4 different occurrences of the PRIFIL4CLB. All the signals involved are grouped into words consisting of 4 4-bit words.

```
⊢ ∀ hiReq genReq req.
     PRIORITY ((hiReq,genReq),req) =
     (FOR i ::(TO (BSIGLEN req)).
       PRIFIL4CLB ((SBITS i hiReq,SBITS i genReq),SBITS i req))
```

```
⊢ ∀ hiReq genReq req.
     PRIORITY_SIMPL ((hiReq,genReq),req) =
     (FOR i ::(TO (BSIGLEN req)).
       PRIFIL4CLB_SPEC ((SBITS i hiReq,SBITS i genReq),SBITS i req))
```

Qudos HDL

```
PriFilter[0-3] := PRIFIL4CLB (
                     hiReq[0-3], hiReq[4-7], hiReq[8-11], hiReq[12-15],
                     genReq[0-3], genReq[4-7], genReq [8-11], genReq[12-15],
                     req[0-3], req[4-7], req[8-11], req[12-15]);
```

3.24.3 The Correctness Statement

```
⊢ FOR hiReq genReq req ::(PSIG2LEN 4 4).
     PRIORITY ((hiReq,genReq),req) ⊃ PRIORITY_SPEC ((hiReq,genReq),req)
```

3.25 HIREQ

3.25.1 The Behavioural Specification

`HIREQ` outputs a 4-bit position vector `hiReq` indicating which output port, if any, the input port under consideration is making an active high priority request for. The behavioural specification takes as arguments an active bit `act` and a priority bit `pri`. If either are false, then the output port requested is ignored. The request is given in the form of two bits `r1` and `r2`, giving the binary encoding for one of four output ports. Two further signals, `r1Bar` and `r2Bar`, are intended to hold the negation of these bits. It has no delay. It is defined in terms of the function `HiReq`.

```
⊢ ∀ act pri r1 r2 r1Bar r2Bar.
     HiReq act pri r1 r2 r1Bar r2Bar =
     WORD
       [act ∧ pri ∧ r1 ∧ r2; act ∧ pri ∧ r1 ∧ r2Bar;
        act ∧ pri ∧ r1Bar ∧ r2; act ∧ pri ∧ r1Bar ∧ r2Bar]
```

```
⊢ ∀ act pri r1 r2 r1Bar r2Bar hiReq.
     HIREQ_SPEC ((act,pri,r1,r2,r1Bar,r2Bar),hiReq) =
     (∀ t. hiReq t = HiReq (act t) (pri t) (r1 t) (r2 t) (r1Bar t) (r2Bar t))
```

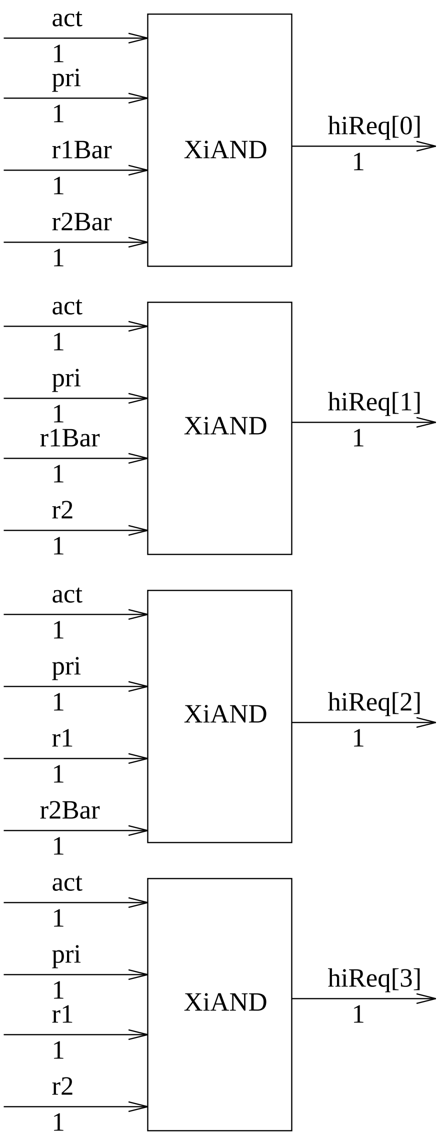

Figure 15: The Implementation of `HIREQ`

3.25.2 The Structural Specification

`HIREQ` gives an extra level of hierarchy over the HDL version.

```
⊢ ∀ act pri r1 r2 r1Bar r2Bar hiReq.
     HIREQ ((act,pri,r1,r2,r1Bar,r2Bar),hiReq) =
     XiAND (SWORD [act; pri; r1Bar; r2Bar],SBIT 0 hiReq) ∧
     XiAND (SWORD [act; pri; r1Bar; r2],SBIT 1 hiReq) ∧
     XiAND (SWORD [act; pri; r1; r2Bar],SBIT 2 hiReq) ∧
     XiAND (SWORD [act; pri; r1; r2],SBIT 3 hiReq)
```

Qudos HDL

```
AndHiReq[0] := XiAND4 (act, pri, r1Bar, r2Bar, hiReq[0]);
AndHiReq[1] := XiAND4 (act, pri, r1Bar, r2, hiReq[1]);
AndHiReq[2] := XiAND4 (act, pri, r1, r2Bar, hiReq[2]);
AndHiReq[3] := XiAND4 (act, pri, r1, r2, hiReq[3]);
```

3.25.3 The Correctness Statement

```
⊢ ∀ act pri r1 r2 r1Bar r2Bar.
     FOR hiReq ::(PSIGLEN 4).
       HIREQ ((act,pri,r1,r2,r1Bar,r2Bar),hiReq) ⊃
       HIREQ_SPEC ((act,pri,r1,r2,r1Bar,r2Bar),hiReq)
```

3.26 GENREQ

3.26.1 The Behavioural Specification

`GENREQ` outputs a 4-bit position vector, `genReq` indicating which output port, if any, the input port under consideration is making an active request for. It takes as arguments an active bit `act` and a priority bit `pri`. If either are false, then the output port requested is ignored. The request is given in the form of two bits `r1` and `r2` giving the binary encoding for one of four output ports. Two further signals, `r1Bar` and `r2Bar`, are intended to hold the negation of these bits. It has no delay. It is defined in terms of the function `GenReq`.

```
⊢ ∀ act r1 r2 r1Bar r2Bar.
     GenReq act r1 r2 r1Bar r2Bar =
     WORD
       [act ∧ r1 ∧ r2; act ∧ r1 ∧ r2Bar; act ∧ r1Bar ∧ r2;
        act ∧ r1Bar ∧ r2Bar]
```

```
⊢ ∀ act r1 r2 r1Bar r2Bar genReq.
     GENREQ_SPEC ((act,r1,r2,r1Bar,r2Bar),genReq) =
     (∀ t. genReq t = GenReq (act t) (r1 t) (r2 t) (r1Bar t) (r2Bar t))
```

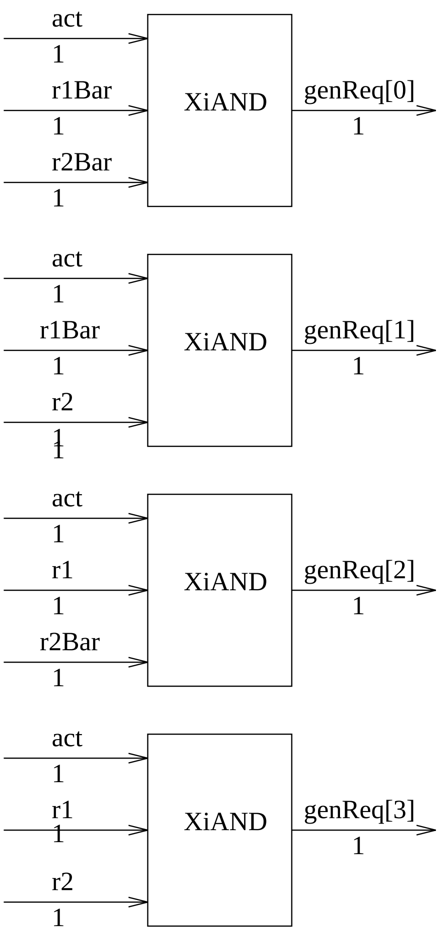

Figure 16: The Implementation of GENREQ

3.26.2 The Structural Specification

GENREQ gives an extra level of hierarchy over the HDL version.

```
⊢ ∀ act r1 r2 r1Bar r2Bar genReq.
    GENREQ ((act,r1,r2,r1Bar,r2Bar),genReq) =
    XiAND (SWORD [act; r1Bar; r2Bar],SBIT 0 genReq) ∧
    XiAND (SWORD [act; r1Bar; r2],SBIT 1 genReq) ∧
    XiAND (SWORD [act; r1; r2Bar],SBIT 2 genReq) ∧
    XiAND (SWORD [act; r1; r2],SBIT 3 genReq)
```

Qudos HDL

```
AndGenReq[0] := XiAND3 (act, r1Bar, r2Bar, genReq[0]);
AndGenReq[1] := XiAND3 (act, r1Bar, r2, genReq[1]);
AndGenReq[2] := XiAND3 (act, r1, r2Bar, genReq[2]);
```

```
AndGenReq[3] := XiAND3 (act, r1, r2, genReq[3]);
```

3.26.3 The Correctness Statement

```
⊢ ∀ act r1 r2 r1Bar r2Bar.
    FOR genReq ::(PSIGLEN 4).
      GENREQ ((act,r1,r2,r1Bar,r2Bar),genReq) ⊃
      GENREQ_SPEC ((act,r1,r2,r1Bar,r2Bar),genReq)
```

3.27 DECODE

3.27.1 The Behavioural Specification

DECODE converts the active, act, priority, pri, and route route signals of an input port into 4-bit position vectors genReq and hiReq. They indicate the output port requested, if any, and the output port requested at high priority, if any, respectively.

act and pri are boolean signals which indicate whether the input port under consideration is making an active request, and if so, whether it is of high priority, respectively. The 2-bit word route indicates in binary the number of the output port it is requesting.

genReq gives the general requests. It has one position for each output port. A T in the position of an output port indicates that it is being requested by the input port.

hiReq gives the high priority requests. It also has one position for each output port. A T in the position of an output port indicates that it is being requested at high priority by the input port.

The behavioural specification is defined in terms of the functions DecoderPriorities and DecoderRequests given earlier.

```
⊢ ∀ act pri route hiReq genReq.
    DECODE_SPEC ((act,pri,route),hiReq,genReq) =
    (∀ t. hiReq t = DecoderPriorities (act t) (pri t) (route t)) ∧
    (∀ t. genReq t = DecoderRequests (act t) (route t))
```

3.27.2 The Structural Specification

DECODE corresponds to the HDL module of the same name. The hardware for calculating the hiReq and genReq signals are placed in separate modules. The route information is given as a 2-bit word rather than as two booleans. The bits are first extracted from this word.

```
⊢ ∀ act pri route hiReq genReq.
    DECODE ((act,pri,route),hiReq,genReq) =
    (LOCAL r1 r2 r1Bar r2Bar.
      (r1 = SBIT 1 route) ∧
      (r2 = SBIT 0 route) ∧
      XiINV (r1,r1Bar) ∧
      XiINV (r2,r2Bar) ∧
      HIREQ ((act,pri,r1,r2,r1Bar,r2Bar),hiReq) ∧
      GENREQ ((act,r1,r2,r1Bar,r2Bar),genReq))
```

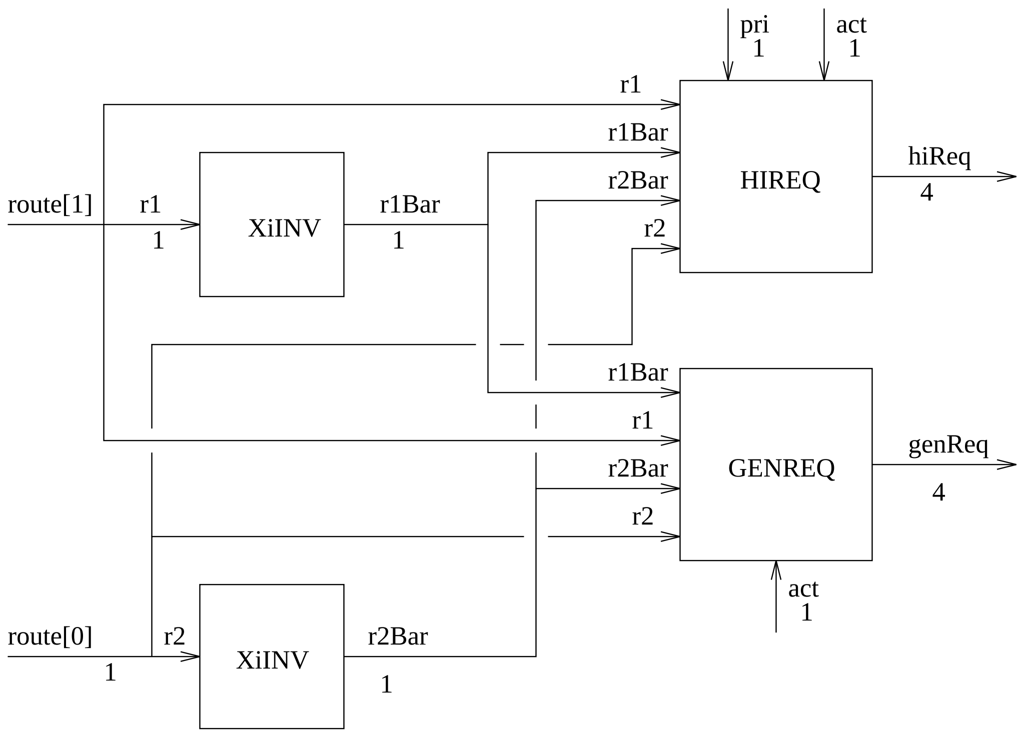

Figure 17: The Implementation of `DECODE`

```
⊢ ∀ act pri route hiReq genReq.
    DECODE_SIMPL ((act,pri,route),hiReq,genReq) =
    (LOCAL r1 r2 r1Bar r2Bar.
      (r1 = SBIT 1 route) ∧
      (r2 = SBIT 0 route) ∧
      XiINV (r1,r1Bar) ∧
      XiINV (r2,r2Bar) ∧
      HIREQ_SPEC ((act,pri,r1,r2,r1Bar,r2Bar),hiReq) ∧
      GENREQ_SPEC ((act,r1,r2,r1Bar,r2Bar),genReq))
```

Qudos HDL

```
DEF DECODE (act, pri, r1, r2: IN; hiReq[0..3], genReq[0..3]: IO);

r1Bar, r2Bar: IO;
BEGIN
    InvR1 := XiINV (r1, r1Bar);
    InvR2 := XiINV (r2, r2Bar);

    AndHiReq[0] := XiAND4 (act, pri, r1Bar, r2Bar, hiReq[0]);
    AndHiReq[1] := XiAND4 (act, pri, r1Bar, r2, hiReq[1]);
    AndHiReq[2] := XiAND4 (act, pri, r1, r2Bar, hiReq[2]);
    AndHiReq[3] := XiAND4 (act, pri, r1, r2, hiReq[3]);


    AndGenReq[0] := XiAND3 (act, r1Bar, r2Bar, genReq[0]);
    AndGenReq[1] := XiAND3 (act, r1Bar, r2, genReq[1]);
    AndGenReq[2] := XiAND3 (act, r1, r2Bar, genReq[2]);
    AndGenReq[3] := XiAND3 (act, r1, r2, genReq[3]);
END;
```

3.27.3 The Correctness Statement

```
⊢ FOR route ::(PSIGLEN 2).
    FOR hiReq genReq ::(PSIGLEN 4).
      DECODE ((act,pri,route),hiReq,genReq) ⊃
      DECODE_SPEC ((act,pri,route),hiReq,genReq)
```

3.28 DECODE_N

3.28.1 The Behavioural Specification

`DECODE_N` is the top level specification of the Decoder block in the fabric. It converts active, priority and request signals from each input port into two position vectors for each input port, `genReq` and `hiReq`. They indicate respectively the output port requested, if any, and the output port requested at high priority, if any, for each input.

The `active` and `priority` input words contain one signal for each input port. They indicate whether it was making an active request, and whether it is of high priority, respectively. `route` consists of a word for each input port indicating the number of the output port it is requesting.

genReq contains one word for each input port giving the general requests. Each word has one bit per output port. A T in the position of an output port indicates that it is being requested by that input port. Similarly, the output hiReq contains one word for each input port giving the high priority requests. Each input port is considered in isolation. The specification is defined in terms of the functions DecoderPriorities and DecoderRequests given earlier.

```
⊢ ∀ active priority route hiReq genReq.
     DECODE_N_SPEC ((active,priority,route),hiReq,genReq) =
     (∀ t.
       hiReq t =
       MKW (SIGLEN hiReq)
         (λ n.
           DecoderPriorities (BIT n (active t)) (BIT n (priority t))
             (BIT n (route t)))) ∧
     (∀ t.
       genReq t =
       MKW (SIGLEN genReq)
         (λ n. DecoderRequests (BIT n (active t)) (BIT n (route t))))
```

3.28.2 The Structural Specification

DECODE_N gives an extra level of hierarchy over the HDL version. It describes the 4 different occurrences of DECODE. The arguments are all structured as words of 4 elements. The active, priority and route information has been split into separate word arguments, rather than just corresponding to positions within d.

```
⊢ ∀ active priority route hiReq genReq.
     DECODE_N ((active,priority,route),hiReq,genReq) =
     (FOR i ::(TO (SIGLEN hiReq)).
       DECODE
         ((SBIT i active,SBIT i priority,SBIT i route),
          SBIT i hiReq,
          SBIT i genReq))
```

```
⊢ ∀ active priority route hiReq genReq.
     DECODE_N_SIMPL ((active,priority,route),hiReq,genReq) =
     (FOR i ::(TO (SIGLEN hiReq)).
       DECODE_SPEC
         ((SBIT i active,SBIT i priority,SBIT i route),
          SBIT i hiReq,
          SBIT i genReq))
```

Qudos HDL

```
Decode[0] := DECODE (d[0..3],   hiReq[0..3],   genReq[0..3]);
Decode[1] := DECODE (d[8..11],  hiReq[4..7],   genReq[4..7]);
Decode[2] := DECODE (d[16..19], hiReq[8..11],  genReq[8..11]);
Decode[3] := DECODE (d[24..27], hiReq[12..15], genReq[12..15]);
```

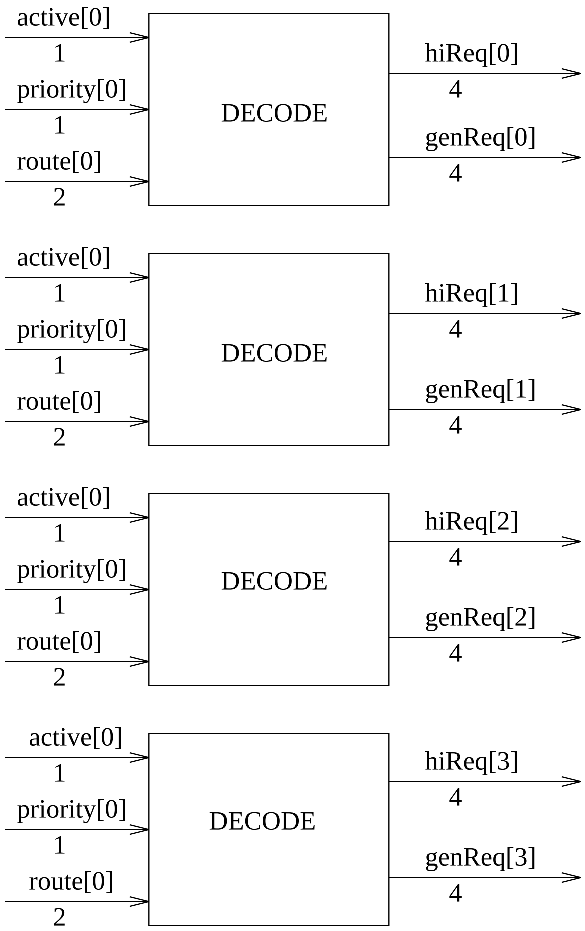

Figure 18: The Implementation of `DECODE_N`

3.28.3 The Correctness Statement

```
⊢ FOR active priority ::(PSIGLEN 4).
     FOR route ::(PSIG2LEN 4 2).
       FOR hiReq genReq ::(PSIG2LEN 4 4).
         DECODE_N ((active,priority,route),hiReq,genReq) ⊃
         DECODE_N_SPEC ((active,priority,route),hiReq,genReq)
```

3.29 ARB_YEL

3.29.1 The Behavioural Specification

ARB_YEL determines the inputs to the JK flip-flop for the low bit of the grant signal from the arbiter. It is defined using the function ArbYel.

```
⊢ ∀ x xBar reqA reqB reqC reqD.
     ArbYel x xBar reqA reqB reqC reqD =
     (x ∨ ~ reqB) ∧ reqC ∨ (xBar ∨ ~ reqA) ∧ reqD
```

```
⊢ ∀ x xBar reqA reqB reqC reqD yjk.
     ARB_YEL_SPEC ((x,xBar,reqA,reqB,reqC,reqD),yjk) =
     (∀ t. yjk t = ArbYel (x t) (xBar t) (reqA t) (reqB t) (reqC t) (reqD t))
```

3.29.2 The Structural Specification

ARB_YEL gives an extra level of hierarchy over the HDL version. The Jy and Ky CLB's of the arbiter are essentially the same. The only differences are that they operate on different bits, and that one uses an inverted x signal. The difference is thus only in the arguments supplied rather than the structure of the hardware. ARB_YEL gives a single description for them.

```
⊢ ∀ x xBar reqA reqB reqC reqD yjk.
     ARB_YEL ((x,xBar,reqA,reqB,reqC,reqD),yjk) =
     (LOCAL reqABar reqBBar term0 term1.
       XiINV (reqA,reqABar) ∧
       XiINV (reqB,reqBBar) ∧
       AO ((term0,reqC,term1,reqD),yjk) ∧
       XiOR (SWORD [x; reqBBar],term0) ∧
       XiOR (SWORD [xBar; reqABar],term1))
```

Qudos HDL

```
(* Jy *)

InvReq0JyCLB := XiINV (req[0], reqBarJyCLB0);
InvReq2JyCLB := XiINV (req[2], reqBarJyCLB2);

AoJy := AO (jyTerm[0], req[3], jyTerm[1], req[1], jy);
```

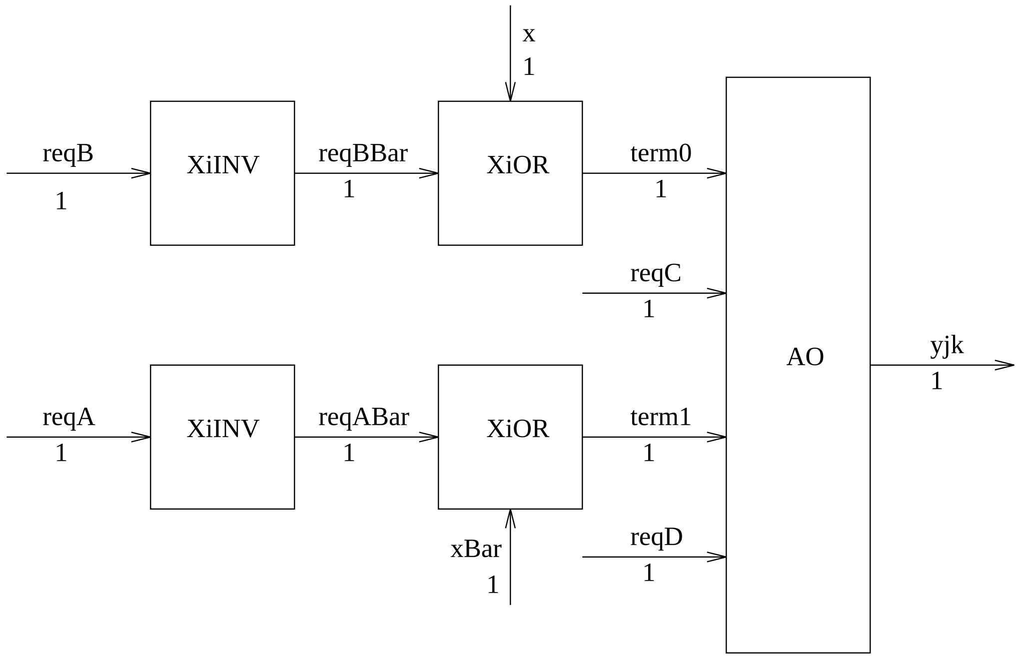

Figure 19: The Implementation of ARB_YEL

```
OrTermJy[0] := XiOR2(x, reqBarJyCLB2, jyTerm[0]);
OrTermJy[1] := XiOR2(xBarJyCLB, reqBarJyCLB0, jyTerm[1]);

(* Ky *)

InvReq1KyCLB := XiINV (req[1], reqBarKyCLB1);
InvReq3KyCLB := XiINV (req[3], reqBarKyCLB3);

AoKy := AO (kyTerm[0], req[2], kyTerm[1], req[0], ky);

OrTermKy[0] := XiOR2(xBarKyCLB, reqBarKyCLB1, kyTerm[0]);
OrTermKy[1] := XiOR2(x, reqBarKyCLB3, kyTerm[1]);
```

3.29.3 The Correctness Statement

```
⊢ ∀ x xBar reqA reqB reqC reqD yjk.
     ARB_YEL ((x,xBar,reqA,reqB,reqC,reqD),yjk) ⊃
     ARB_YEL_SPEC ((x,xBar,reqA,reqB,reqC,reqD),yjk)
```

3.30 ARB_XEL

3.30.1 The Behavioural Specification

ARB_XEL determines the inputs to the JK flip-flop for the high bit of the grant signal from the

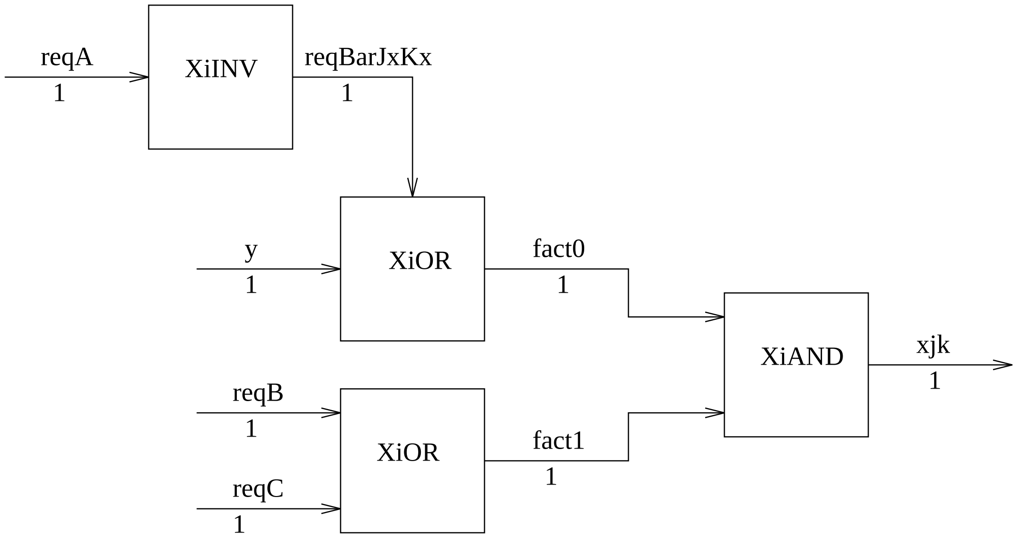

Figure 20: The Implementation of ARB_XEL

arbiter.

```
⊢ ∀ reqA y reqB reqC. ArbXel reqA y reqB reqC = (y ∨ ∼ reqA) ∧ (reqB ∨ reqC)
```

```
⊢ ∀ reqA y reqB reqC xjk.
     ARB_XEL_SPEC ((reqA,y,reqB,reqC),xjk) =
     (∀ t. xjk t = ArbXel (reqA t) (y t) (reqB t) (reqC t))
```

3.30.2 The Structural Specification

ARB_XEL gives an extra level of hierarchy over the HDL version. The Jx and Kx part of the arbiter are essentially the same. The only difference is that they operate on different bits. ARBX_EL describes the common hardware.

```
⊢ ∀ reqA y reqB reqC xjk.
     ARB_XEL ((reqA,y,reqB,reqC),xjk) =
     (LOCAL reqBarJxKx fact0 fact1.
       XiINV (reqA,reqBarJxKx) ∧
       XiAND (SWORD [fact0; fact1],xjk) ∧
       XiOR (SWORD [y; reqBarJxKx],fact0) ∧
       XiOR (SWORD [reqB; reqC],fact1))
```

Qudos HDL

```
(* Jx  *)

InvReqJxKxCLB1 := XiINV (req[1], reqBarJxKxCLB1);

AndJx := XiAND2(jxFact[0..1], jx);
OrFactJx[0] := XiOR2(y, reqBarJxKxCLB1, jxFact[0]);
OrFactJx[1] := XiOR2(req[3], req[2], jxFact[1]);

(* Kx  *)

InvReqJxKxCLB3 := XiINV (req[3], reqBarJxKxCLB3);

AndKx := XiAND2(kxFact[0..1], kx);
OrFactKx[0] := XiOR2(y, reqBarJxKxCLB3, kxFact[0]);
OrJactKx[1] := XiOR2(req[0], req[1], kxFact[1]);
```

3.30.3 The Correctness Statement

```
⊢ ∀ y reqA reqB reqC xjk.
     ARB_XEL ((reqA,y,reqB,reqC),xjk) ⊃ ARB_XEL_SPEC ((reqA,y,reqB,reqC),xjk)
```

3.31 K_ARBY

3.31.1 The Behavioural Specification

K_ARBY determines the K input to the JK flip-flop for the low bit of the grant signal from the arbiter.

```
⊢ ∀ x req ky.
     K_ARBY_SPEC ((x,req),ky) =
     (∀ t.
       ky t =
       ArbYel (~ (x t)) (x t) (BIT 3 (req t)) (BIT 1 (req t)) (BIT 2 (req t))
         (BIT 0 (req t)))
```

3.31.2 The Structural Specification

K_ARBY gives an extra level of hierarchy over the HDL version. It combines the inverter with a Y element, passing to it the appropriate bits for a K Y element. Note that xBar and x are inverted.

```
⊢ ∀ x req ky.
     K_ARBY ((x,req),ky) =
     (LOCAL xBar.
       XiINV (x,xBar) ∧
       ARB_YEL ((xBar,x,SBIT 3 req,SBIT 1 req,SBIT 2 req,SBIT 0 req),ky))
```

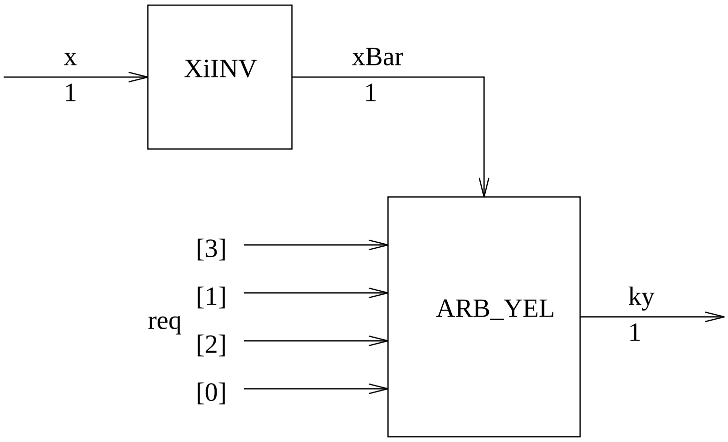

Figure 21: The Implementation of K_ARBY

```
⊢ ∀ x req ky.
     K_ARBY_SIMPL ((x,req),ky) =
     (LOCAL xBar.
       XiINV (x,xBar) ∧
       ARB_YEL_SPEC ((xBar,x,SBIT 3 req,SBIT 1 req,SBIT 2 req,SBIT 0 req),ky))
```

Qudos HDL

```
(* Ky *)

ClbKy := XiCLBMAP5i1o (x, req[0..3], ky);
InvXKyCLB := XiINV (x, xBarKyCLB);
InvReq1KyCLB := XiINV (req[1], reqBarKyCLB1);
InvReq3KyCLB := XiINV (req[3], reqBarKyCLB3);

AoKy := AO (kyTerm[0], req[2], kyTerm[1], req[0], ky);

OrTermKy[0] := XiOR2(xBarKyCLB, reqBarKyCLB1, kyTerm[0]);
OrTermKy[1] := XiOR2(x, reqBarKyCLB3, kyTerm[1]);
```

3.31.3 The Correctness Statement

```
⊢ K_ARBY ((x,req),ky) ⊃ K_ARBY_SPEC ((x,req),ky)
```

3.32 J_ARBY

3.32.1 The Behavioural Specification

J_ARBY determines the J input to the JK flip-flop for the low bit of the grant signal from the arbiter.

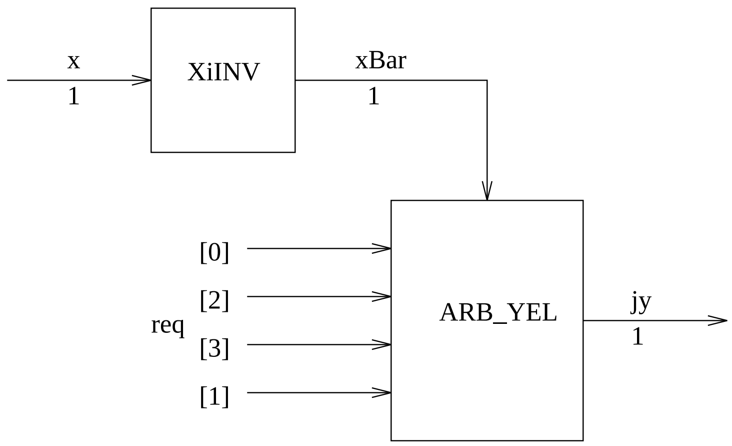

Figure 22: The Implementation of J_ARBY

```
⊢ ∀ x req jy.
     J_ARBY_SPEC ((x,req),jy) =
     (∀ t.
       jy t =
       ArbYel (x t) (~ (x t)) (BIT 0 (req t)) (BIT 2 (req t)) (BIT 3 (req t))
         (BIT 1 (req t)))
```

3.32.2 The Structural Specification

J_ARBY gives an extra level of hierarchy over the HDL version. It combines the inverter with a Y element, passing the appropriate bits for a J Y element.

```
⊢ ∀ x req jy.
     J_ARBY ((x,req),jy) =
     (LOCAL xBar.
       XiINV (x,xBar) ∧
       ARB_YEL ((x,xBar,SBIT 0 req,SBIT 2 req,SBIT 3 req,SBIT 1 req),jy))
```

```
⊢ ∀ x req jy.
     J_ARBY_SIMPL ((x,req),jy) =
     (LOCAL xBar.
       XiINV (x,xBar) ∧
       ARB_YEL_SPEC ((x,xBar,SBIT 0 req,SBIT 2 req,SBIT 3 req,SBIT 1 req),jy))
```

Qudos HDL

```
(* Jy *)

ClbJy := XiCLBMAP5i1o (x, req[0..3], jy);
```

```
InvXJyCLB := XiINV (x, xBarJyCLB);
InvReq0JyCLB := XiINV (req[0], reqBarJyCLB0);
InvReq2JyCLB := XiINV (req[2], reqBarJyCLB2);

AoJy := AO (jyTerm[0], req[3], jyTerm[1], req[1], jy);

OrTermJy[0] := XiOR2(x, reqBarJyCLB2, jyTerm[0]);
OrTermJy[1] := XiOR2(xBarJyCLB, reqBarJyCLB0, jyTerm[1]);
```

3.32.3 The Correctness Statement

```
⊢ J_ARBY ((x,req),ky) ⊃ J_ARBY_SPEC ((x,req),ky)
```

3.33 ARBY

3.33.1 The Behavioural Specification

ARBY determines the J and K inputs to the JK flip-flop for the low bit of the grant signal from the arbiter.

```
⊢ ∀ x req jy ky.
     ARBY_SPEC ((x,req),jy,ky) =
     (∀ t.
       jy t =
       ArbYel (x t) (~ (x t)) (BIT 0 (req t)) (BIT 2 (req t)) (BIT 3 (req t))
         (BIT 1 (req t))) ∧
     (∀ t.
       ky t =
       ArbYel (~ (x t)) (x t) (BIT 3 (req t)) (BIT 1 (req t)) (BIT 2 (req t))
         (BIT 0 (req t)))
```

3.33.2 The Structural Specification

ARBY gives an extra level of hierarchy over the HDL version. It combines the J Y and K Y elements.

```
⊢ ∀ x req jy ky.
     ARBY ((x,req),jy,ky) = J_ARBY ((x,req),jy) ∧ K_ARBY ((x,req),ky)
```

```
⊢ ∀ x req jy ky.
     ARBY_SIMPL ((x,req),jy,ky) =
     J_ARBY_SPEC ((x,req),jy) ∧ K_ARBY_SPEC ((x,req),ky)
```

Qudos HDL

```
(* Jy *)

ClbJy := XiCLBMAP5i1o (x, req[0..3], jy);
```

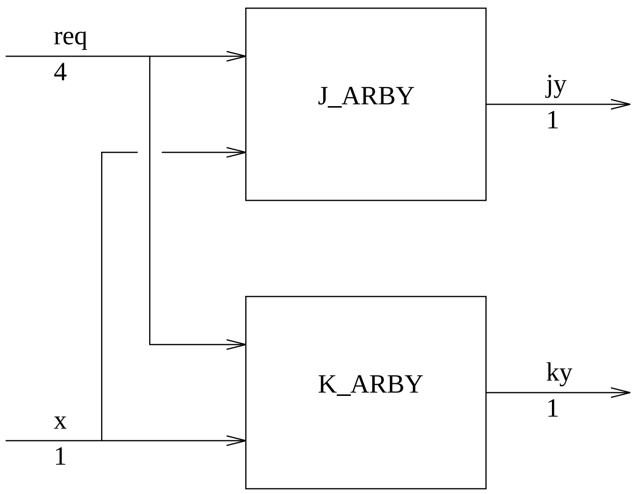

Figure 23: The Implementation of ARBY

```
InvXJyCLB := XiINV (x, xBarJyCLB);
InvReq0JyCLB := XiINV (req[0], reqBarJyCLB0);
InvReq2JyCLB := XiINV (req[2], reqBarJyCLB2);

AoJy := AO (jyTerm[0], req[3], jyTerm[1], req[1], jy);

OrTermJy[0] := XiOR2(x, reqBarJyCLB2, jyTerm[0]);
OrTermJy[1] := XiOR2(xBarJyCLB, reqBarJyCLB0, jyTerm[1]);


(* Ky *)

ClbKy := XiCLBMAP5i1o (x, req[0..3], ky);
InvXKyCLB := XiINV (x, xBarKyCLB);
InvReq1KyCLB := XiINV (req[1], reqBarKyCLB1);
InvReq3KyCLB := XiINV (req[3], reqBarKyCLB3);

AoKy := AO (kyTerm[0], req[2], kyTerm[1], req[0], ky);

OrTermKy[0] := XiOR2(xBarKyCLB, reqBarKyCLB1, kyTerm[0]);
OrTermKy[1] := XiOR2(x, reqBarKyCLB3, kyTerm[1]);


ClbJxKx := XiCLBMAP5i2o(req[0], req[1], req[2], y, req[3], jx, kx);

InvReqJxKxCLB1 := XiINV (req[1], reqBarJxKxCLB1);
InvReqJxKxCLB3 := XiINV (req[3], reqBarJxKxCLB3);
```

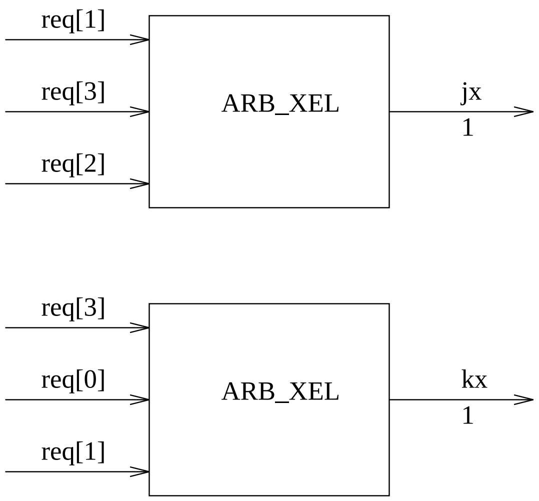

Figure 24: The Implementation of ARBX

3.33.3 The Correctness Statement

```
⊢ ARBY ((x,req),jy,ky) ⊃ ARBY_SPEC ((x,req),jy,ky)
```

3.34 ARBX

3.34.1 The Behavioural Specification

ARBX determines the J and K inputs to the JK flip-flop for the high bit of the grant signal from the arbiter.

```
⊢ ∀ y req jx kx.
     ARBX_SPEC ((y,req),jx,kx) =
     (∀ t.
       jx t = ArbXel (BIT 1 (req t)) (y t) (BIT 3 (req t)) (BIT 2 (req t))) ∧
     (∀ t. kx t = ArbXel (BIT 3 (req t)) (y t) (BIT 0 (req t)) (BIT 1 (req t)))
```

3.34.2 The Structural Specification

ARBX gives an extra level of hierarchy over the HDL version. It combines the J X and K X elements passing the appropriate bits in each case.

```
⊢ ∀ y req jx kx.
     ARBX ((y,req),jx,kx) =
     ARB_XEL ((SBIT 1 req,y,SBIT 3 req,SBIT 2 req),jx) ∧
     ARB_XEL ((SBIT 3 req,y,SBIT 0 req,SBIT 1 req),kx)
```

```
⊢ ∀ y req jx kx.
     ARBX_SIMPL ((y,req),jx,kx) =
     ARB_XEL_SPEC ((SBIT 1 req,y,SBIT 3 req,SBIT 2 req),jx) ∧
     ARB_XEL_SPEC ((SBIT 3 req,y,SBIT 0 req,SBIT 1 req),kx)
```

Qudos HDL

```
ClbJxKx := XiCLBMAP5i2o(req[0], req[1], req[2], y, req[3], jx, kx);

InvReqJxKxCLB1 := XiINV (req[1], reqBarJxKxCLB1);
InvReqJxKxCLB3 := XiINV (req[3], reqBarJxKxCLB3);

(* Jx  *)

AndJx := XiAND2(jxFact[0..1], jx);
OrFactJx[0] := XiOR2(y, reqBarJxKxCLB1, jxFact[0]);
OrFactJx[1] := XiOR2(req[3], req[2], jxFact[1]);

(* Kx  *)

AndKx := XiAND2(kxFact[0..1], kx);
OrFactKx[0] := XiOR2(y, reqBarJxKxCLB3, kxFact[0]);
OrJactKx[1] := XiOR2(req[0], req[1], kxFact[1]);
```

3.34.3 The Correctness Statement

```
⊢ ARBX ((y,req),jx,kx) ⊃ ARBX_SPEC ((y,req),jx,kx)
```

3.35 ARBITER_FF

3.35.1 The Behavioural Specification

ARBITER_FF consists of 2 independent JK flip-flops, each enabled by the routeEnable signal.

```
⊢ ∀ jx kx jy ky routeEnable x y.
     ARBITER_FF_SPEC ((jx,kx,jy,ky,routeEnable),x,y) =
     (∀ t.
       (x (t + 1) = JkE (jx t) (kx t) (x t) (routeEnable t)) ∧
       (y (t + 1) = JkE (jy t) (ky t) (y t) (routeEnable t)))
```

3.35.2 The Structural Specification

ARBITER_FF gives an extra level of hierarchy over the HDL version. It describes the 2-flip flop CLB. The clock signals are not included.

```
⊢ ∀ jx kx jy ky routeEnable x y.
     ARBITER_FF ((jx,kx,jy,ky,routeEnable),x,y) =
     JKFFce ((jx,kx,routeEnable),x) ∧ JKFFce ((jy,ky,routeEnable),y)
```

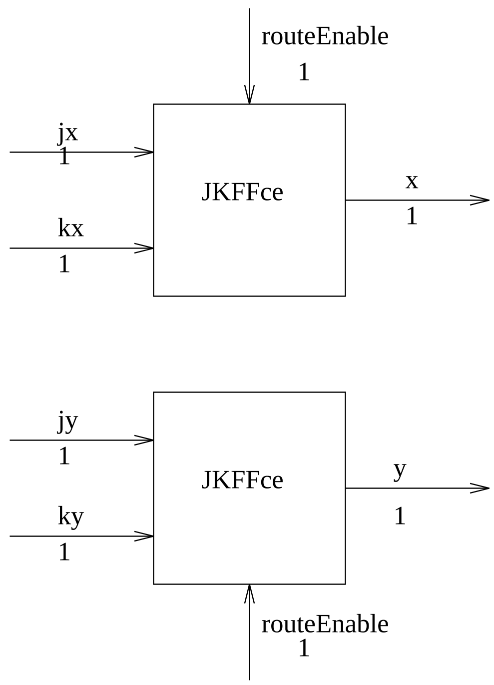

Figure 25: The Implementation of ARBITER_FF

Qudos HDL

```
ClbFF := XiCLBMAP5i2oke (jx, kx, jx, jy, ky, clock, routeEnable, x, y);

FFx := JKFFce (jx, kx, clock, routeEnable, x);
FFy := JKFFce (jy, ky, clock, routeEnable, y);
```

3.35.3 The Correctness Statement

```
⊢ ∀ jx kx jy ky routeEnable x y.
     ARBITER_FF ((jx,kx,jy,ky,routeEnable),x,y) ⊃
     ARBITER_FF_SPEC ((jx,kx,jy,ky,routeEnable),x,y)
```

3.36 OUTDIS

3.36.1 The Behavioural Specification

`OUTDIS` describes the timing diagram for the `OutputDisable` signal for a single output port. At the start of a frame it is set to high, disabling the outputs. If during the frame an active packet arrives for the output, it is set to low for the remainder of the cycle. The frame boundaries are determined by the `frameStart` signal. Outputs are disabled if there are requests from any input when the `routeEnable` signal is activated. The circuit does not operate correctly if a `routeEnable` and `frameStart` signal arrive at the same time. If it were to occur, `TIMING` would act as though it had received a cell in the previous frame, but OUTDIS would act as though the cell was in the next frame. The environment must ensure this does not happen. This condition is part of the specification of the TIMING module.

As with the TIMING module, two specifications are given—one at the frame level and one at the clock level. In the frame level description the "active signal" being high corresponds to there being at least one request and the routeEnable signal being high: (λt. (EXISTSABIT I (req t) ∧ routeEnable t))

```
⊢ ∀ fs routeEnable req outputDisable.
     OUTDIS_SPEC1 ((fs,routeEnable,req),outputDisable) =
     (∀ t.
       outputDisable (t + 1) =
       ((outputDisable t)
        ⇒ (~ (EXISTSABIT I (req t) ∧ routeEnable t))
        |  (fs t)))
```

```
⊢ ∀ fs routeEnable req outputDisable.
     OUTDIS_SPEC2 ((fs,routeEnable,req),outputDisable) =
     (∀ t_start t_active t_end.
       (∀ t. ~ (fs t ∧ routeEnable t)) ⊃
       (AFRAME1 t_start t_active t_end fs
          (λ t. EXISTSABIT I (req t) ∧ routeEnable t) ⊃
        STABLE (t_start + 1) (t_active + 1) outputDisable T ∧
        STABLE (t_active + 1) (t_end + 1) outputDisable F) ∧
       (IFRAME1 t_start t_end fs
          (λ t. EXISTSABIT I (req t) ∧ routeEnable t) ⊃
        STABLE (t_start + 1) (t_end + 1) outputDisable T))
```

3.36.2 The Structural Specification

OUTDIS gives an extra level of hierarchy over the HDL version. It contains the output disable hardware.

```
⊢ ∀ fs routeEnable req outputDisable.
     OUTDIS ((fs,routeEnable,req),outputDisable) =
     (LOCAL anyReq kOut outputEnable.
       XiOR (req,anyReq) ∧
       XiAND (SWORD [anyReq; routeEnable],kOut) ∧
       JKFF ((fs,kOut),outputDisable,outputEnable))
```

Qudos HDL

```
AnyOr := XiOR4(req[0..3], anyReq);
AndKout := XiAND2 (anyReq, routeEnable, kOut);
FFOutDis := JKFF (fs, kOut, clock, outputDisable, outputEnable);
```

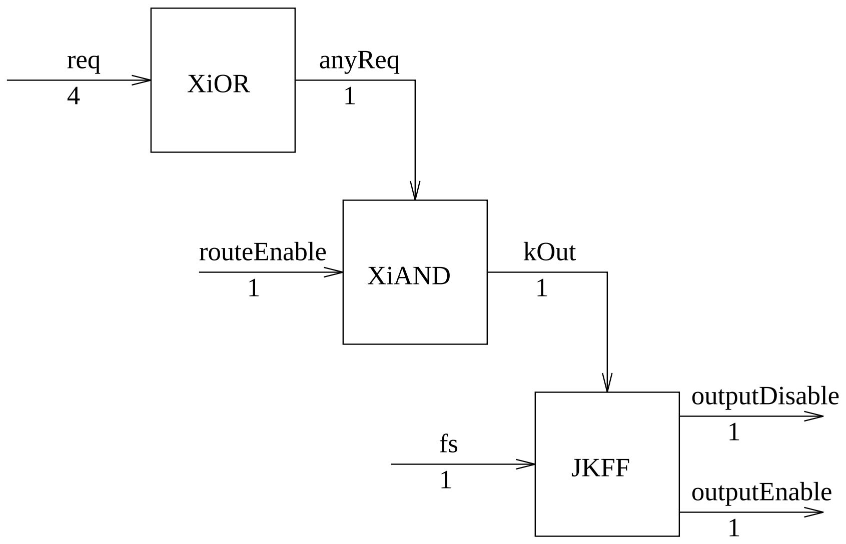

Figure 26: The Implementation of OUTDIS

3.36.3 The Correctness Statement

As with TIMING, OUTDIS is verified in three stages. Firstly, the implementation is verified against the clock level specification. The clock level specification is then verified against the frame level one. Finally the first two theorems are combined to give a theorem relating the implementation to the frame level specification.

```
⊢ ∀ fs routeEnable outputDisable.
    FOR req ::(PSIGLEN (SUC n)).
      OUTDIS ((fs,routeEnable,req),outputDisable) ⊃
      OUTDIS_SPEC1 ((fs,routeEnable,req),outputDisable)
```

```
⊢ ∀ fs routeEnable outputDisable.
    FOR req ::(PSIGLEN (SUC n)).
      OUTDIS_SPEC1 ((fs,routeEnable,req),outputDisable) ⊃
      OUTDIS_SPEC2 ((fs,routeEnable,req),outputDisable)
```

```
⊢ ∀ fs routeEnable outputDisable.
    FOR req ::(PSIGLEN (SUC n)).
      OUTDIS ((fs,routeEnable,req),outputDisable) ⊃
      OUTDIS_SPEC2 ((fs,routeEnable,req),outputDisable)
```

Of these proofs, only the second is at all difficult. It is split into three cases: one for an inactive frame, one for an active frame prior to the active signal and one for an active frame after the active signal.

The first two cases are essentially the same. We must prove that through some initial period in which there is no active signal, `outputDisable` remains true. We know that `frameStart` is high and therefore that `routeEnable` is low at the start of the frame. From `OUTDIS_SPEC1` we can see that at the subsequent time `outputDisable` must be high. Once it is high we can disregard the value of `frameStart`. It will remain high while there is no request at the same time as a `routeEnable` signal. However this is the same as saying no active signal occurs. Thus for an inactive frame, `outputDisable` will remain high until the end of the frame. For an inactive frame it will remain high until the active time.

The third case requires similar reasoning. We know from the above that `outputDisable` will be high at the active time. At this time, there is an active signal, so it will go low on the following cycle. Throughout the remainder of the frame, only the `frameStart` signal is therefore relevant. Consequently `outputDisable` will remain low.

The assumption that a `routeEnable` signal does not arrive at the same time as the `frameStart` signal is required to ensure that the `routeEnable` is not ignored. After generating such a `routeEnable` signal, the `TIMING` unit remains in the state where it is waiting for a cell. It thus treats the header that generated the `routeEnable` as if it had been part of a cell on the previous cycle. `OUTDIS` treats header as if it was part of a cell in the frame which has just started and consequently will enable outputs.

The timing circuitry ensures that the assumption holds provided header bytes do not arrive too close to the `frameStart` signal at the outside of the element.

3.37 ARBITER_XY

3.37.1 The Behavioural Specification

`ARBITER_XY` computes the values of the two bits, `x` and `y`, of the grant signal. If `routeEnable` was not set, they are left unchanged. Otherwise they depend on the previous `x` and `y` values and the requests made. The specification is given in terms of the functions `ArbX` and `ArbY`.

`ArbX` specifies the value of the most significant bit of the grant signal in terms of the values on the previous cycle. If `routeEnable` was not set, it is left unchanged. If it was previously set its value depends on the negated `y` value of the previous cycle and the requests made. Otherwise it depends on the un-negated `y` value and the requests made.

`ArbY` specifies the value of the least significant bit of the grant signal in terms of the values on the previous cycle. If `routeEnable` was not set, it is left unchanged. Otherwise it depends on the previous `x` and `y` values and the requests made.

```
⊢ ∀ ltReq routeEnable y x.
     ArbX ltReq routeEnable y x =
     (routeEnable
      ⇒ (x
          ⇒ (∼ y ∧ BIT 3 ltReq ∨ ∼ (BIT 0 ltReq) ∧ ∼ (BIT 1 ltReq))
          |  ((y ∨ ∼ (BIT 1 ltReq)) ∧ (BIT 3 ltReq ∨ BIT 2 ltReq)))
      |  x)
```

```
⊢ ∀ ltReq routeEnable y x.
     ArbY ltReq routeEnable y x =
     (routeEnable
      ⇒ (y
          ⇒ ((x ∧ BIT 1 ltReq ∨ ~ (BIT 2 ltReq)) ∧
               (~ x ∧ BIT 3 ltReq ∨ ~ (BIT 0 ltReq)))
          |  ((x ∨ ~ (BIT 2 ltReq)) ∧ BIT 3 ltReq ∨
              (~ x ∨ ~ (BIT 0 ltReq)) ∧ BIT 1 ltReq))
      |  y)
```

```
⊢ ∀ ltReq routeEnable x y.
     ARBITER_XY_SPEC ((ltReq,routeEnable),x,y) =
     (∀ t.
       (x (t + 1) = ArbX (ltReq t) (routeEnable t) (y t) (x t)) ∧
       (y (t + 1) = ArbY (ltReq t) (routeEnable t) (y t) (x t)))
```

3.37.2 The Structural Specification

`ARBITER_XY` gives an additional level of hierarchy over the HDL. It is the part of the arbiter that produces the two signals `x` and `y` that make up the grant signal for a single output. It has been structured into separate parts. One definition gives the flip-flop CLB, one the Y hardware and one the X hardware. The clock signal is omitted.

```
⊢ ∀ ltReq routeEnable x y.
     ARBITER_XY ((ltReq,routeEnable),x,y) =
     (LOCAL jx kx jy ky.
       ARBITER_FF ((jx,kx,jy,ky,routeEnable),x,y) ∧
       ARBY ((x,ltReq),jy,ky) ∧
       ARBX ((y,ltReq),jx,kx))
```

```
⊢ ∀ ltReq routeEnable x y.
     ARBITER_XY_SIMPL ((ltReq,routeEnable),x,y) =
     (LOCAL jx kx jy ky.
       ARBITER_FF_SPEC ((jx,kx,jy,ky,routeEnable),x,y) ∧
       ARBY_SPEC ((x,ltReq),jy,ky) ∧
       ARBX_SPEC ((y,ltReq),jx,kx))
```

Qudos HDL

```
(* The State Flip Flops --- XY is last Grant made  *)

ClbFF := XiCLBMAP5i2oke (jx, kx, jx, jy, ky, clock, routeEnable, x, y);

FFx := JKFFce (jx, kx, clock, routeEnable, x);
FFy := JKFFce (jy, ky, clock, routeEnable, y);
```

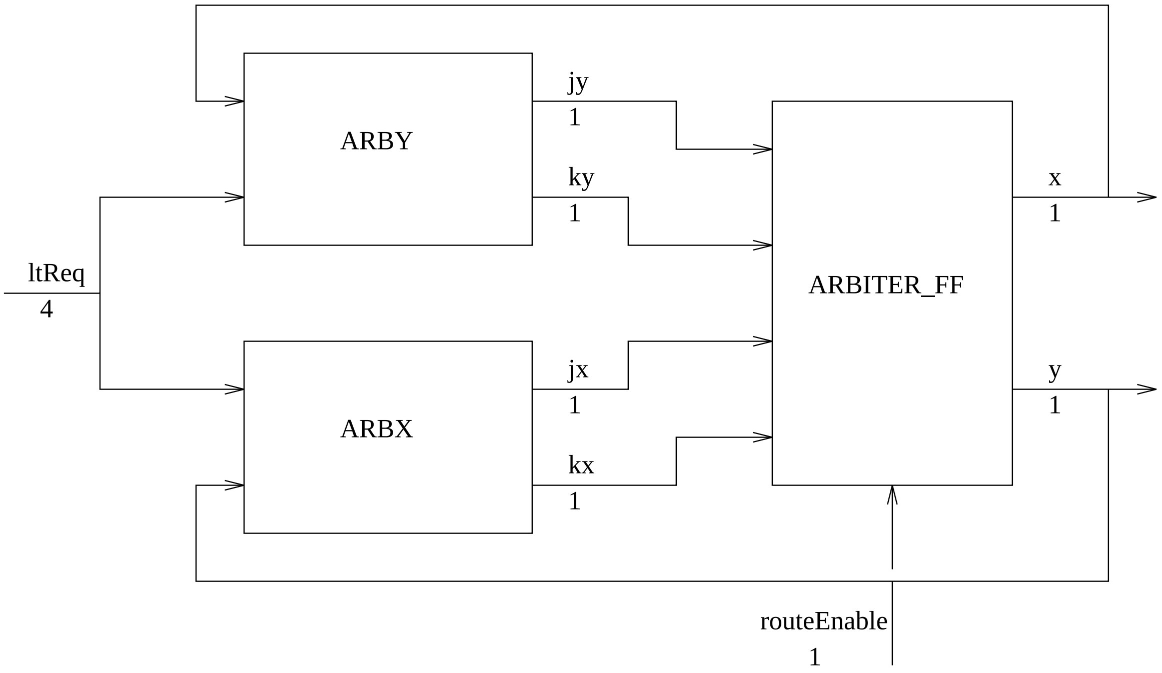

Figure 27: The Implementation of ARBITER_XY

```
(* Jy *)

ClbJy := XiCLBMAP5i1o (x, req[0..3], jy);
InvXJyCLB := XiINV (x, xBarJyCLB);
InvReq0JyCLB := XiINV (req[0], reqBarJyCLB0);
InvReq2JyCLB := XiINV (req[2], reqBarJyCLB2);

AoJy := AO (jyTerm[0], req[3], jyTerm[1], req[1], jy);

OrTermJy[0] := XiOR2(x, reqBarJyCLB2, jyTerm[0]);
OrTermJy[1] := XiOR2(xBarJyCLB, reqBarJyCLB0, jyTerm[1]);


(* Ky *)

ClbKy := XiCLBMAP5i1o (x, req[0..3], ky);
InvXKyCLB := XiINV (x, xBarKyCLB);
InvReq1KyCLB := XiINV (req[1], reqBarKyCLB1);
InvReq3KyCLB := XiINV (req[3], reqBarKyCLB3);

AoKy := AO (kyTerm[0], req[2], kyTerm[1], req[0], ky);

OrTermKy[0] := XiOR2(xBarKyCLB, reqBarKyCLB1, kyTerm[0]);
OrTermKy[1] := XiOR2(x, reqBarKyCLB3, kyTerm[1]);
```

```
ClbJxKx := XiCLBMAP5i2o(req[0], req[1], req[2], y, req[3], jx, kx);

InvReqJxKxCLB1 := XiINV (req[1], reqBarJxKxCLB1);
InvReqJxKxCLB3 := XiINV (req[3], reqBarJxKxCLB3);

(* Jx  *)

AndJx := XiAND2(jxFact[0..1], jx);
OrFactJx[0] := XiOR2(y, reqBarJxKxCLB1, jxFact[0]);
OrFactJx[1] := XiOR2(req[3], req[2], jxFact[1]);

(* Kx  *)

AndKx := XiAND2(kxFact[0..1], kx);
OrFactKx[0] := XiOR2(y, reqBarJxKxCLB3, kxFact[0]);
OrJactKx[1] := XiOR2(req[0], req[1], kxFact[1]);
```

3.37.3 The Correctness Statement

```
⊢ ARBITER_XY ((ltReq,routeEnable),x,y) ⊃
   ARBITER_XY_SPEC ((ltReq,routeEnable),x,y)
```

3.38 ARBITER

3.38.1 The Behavioural Specification

ARBITER performs round-robin arbitration for a single output port. It takes as input a position vector, ltReq, indicating which inputs are making requests for the output, an enable signal routeEnable which indicates when within a frame an arbitration decision should be made, and the framing signal frameStart. It outputs a grant signal which indicates the input whose request is currently accepted and an outputDisable signal which indicates whether the grant signal is currently valid. If it is not it should be ignored and the outputs disabled. This occurs if no input port requests the output port during a frame.

The circuit does not operate correctly if a routeEnable and frameStart signal arrive at the same time. The environment must ensure this does not happen. This condition is part of the specification of the TIMING module. It is an explicit assumption in the behavioural specification of the ARBITER. It is required so that the circuitry resets itself correctly at the start of a frame. Further details are given in the description of OUTDIS.

The behaviour can be split into two parts: that for an active frame and that for an inactive frame. For the arbiter the active signal is the combination of the routeEnable signal and there being a high on at least one bit of ltReq (ie this output port is being requested). This definition of the active signal means that the output port for each separate arbiter will have an active cycle only if it is being requested. Thus some arbiters can be active whilst others are inactive.

The inactive frame behaviour is the simplest. For frame start and end times ts and te respectively, outputDisable should remain true from ts+1 to te+1, and the value on grant should remain unchanged from `ts+2` to `te+2`.

For an active frame with active time `ta`, up until `ta+1` the behaviour is as for an inactive frame. In the remainder of the frame however, `outputDisable` becomes and remains false, and `grant` takes on the value of the newly selected input port for this output port. One will be selected because at least one must be making a request for the cycle to be considered an active cycle for this output port.

```
⊢ ∀ ltReq routeEnable frameStart grant outputDisable.
     ARBITER_SPEC ((ltReq,routeEnable,frameStart),grant,outputDisable) =
     (∀ t_start t_active t_end.
       (∀ t. ~ (frameStart t ∧ routeEnable t)) ⊃
       (STABLE (t_active + 1) (t_end + 1) routeEnable F ⊃
        AFRAME1 t_start t_active t_end frameStart
          (λ t. EXISTSABIT I (ltReq t) ∧ routeEnable t) ⊃
        STABLE (t_start + 2) (t_active + 1) grant (grant (t_start + 1)) ∧
        STABLE (t_active + 1) (t_end + 2) grant
          (GrantForOut (ltReq t_active) (grant t_active)) ∧
        STABLE (t_start + 1) (t_active + 1) outputDisable T ∧
        STABLE (t_active + 1) (t_end + 1) outputDisable F) ∧
       (IFRAME1 t_start t_end frameStart
          (λ t. EXISTSABIT I (ltReq t) ∧ routeEnable t) ⊃
        STABLE (t_start + 2) (t_end + 2) grant (grant (t_start + 1)) ∧
        STABLE (t_start + 1) (t_end + 1) outputDisable T))
```

The round robin arbitration decision making process for a single output port is described by `GrantForOut`. The requests are in the form of a position vector `req`, with one boolean bit for each input port, indicating whether it is making a request or not. The last input port granted a request is given as a binary-encoded word `last`. It is converted to a number, used to compute the successful input, and the result converted back to a binary-encoded word.

```
⊢ ∀ req last.
     GrantForOut req last =
     (let suc_inp = SuccessfulInput (BNVAL last) req
      in
      ((suc_inp = NO_RESULT)
       ⇒ last
       |  (NBWORD (WORDLEN last) (ResultOf suc_inp))))
```

3.38.2 The Structural Specification

`ARBITER` corresponds to the HDL of the same name. It has been structured into separate parts. One definition gives the X and Y hardware and one for the output-disable hardware. The x and y inputs are grouped into a single 2-bit word, `grant`. The clock signal is omitted.

```
⊢ ∀ ltReq routeEnable frameStart grant outputDisable.
     ARBITER ((ltReq,routeEnable,frameStart),grant,outputDisable) =
     ARBITER_XY ((ltReq,routeEnable),SBIT 1 grant,SBIT 0 grant) ∧
     OUTDIS ((frameStart,routeEnable,ltReq),outputDisable)
```

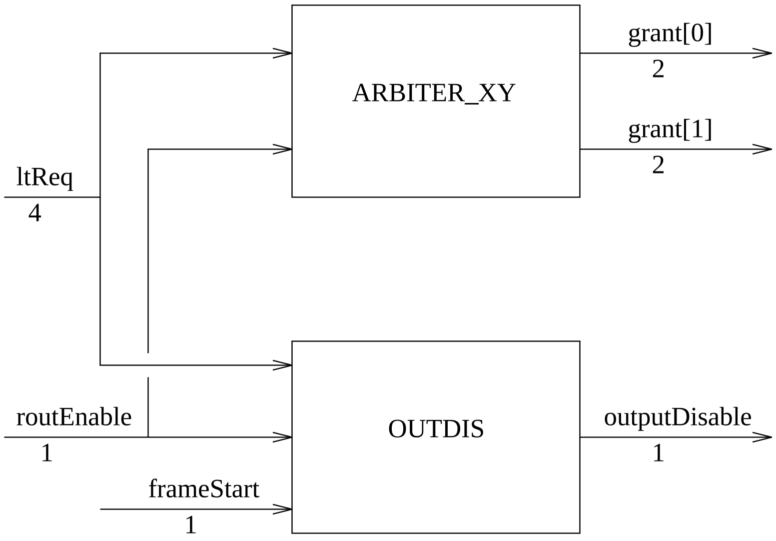

Figure 28: The Implementation of `ARBITER`

```
⊢ ∀ ltReq routeEnable frameStart grant outputDisable.
     ARBITER_SIMPL ((ltReq,routeEnable,frameStart),grant,outputDisable) =
     ARBITER_XY_SPEC ((ltReq,routeEnable),SBIT 1 grant,SBIT 0 grant) ∧
     OUTDIS_SPEC2 ((frameStart,routeEnable,ltReq),outputDisable)
```

Qudos HDL

```
DEF ARBITER(req[0..3], clock, routeEnable, fs: IN; x, y, outputDisable: IO);

jx,kx, jy,ky, jyTerm[0..1], kyTerm[0..1],
jxFact[0..1], kxFact[0..1],
xBarJyCLB, reqBarJyCLB0, reqBarJyCLB2,
xBarKyCLB, reqBarKyCLB1, reqBarKyCLB3,
reqBarJxKxCLB1, reqBarJxKxCLB3,
anyReq, kOut, outputEnable : IO

BEGIN


  (* The State Flip Flops --- XY is last Grant made  *)

  ClbFF := XiCLBMAP5i2oke (jx, kx, jx, jy, ky, clock, routeEnable, x, y);

  FFx := JKFFce (jx, kx, clock, routeEnable, x);
  FFy := JKFFce (jy, ky, clock, routeEnable, y);


  (* Jy *)
```

```
  ClbJy := XiCLBMAP5i1o (x, req[0..3], jy);
  InvXJyCLB := XiINV (x, xBarJyCLB);
  InvReq0JyCLB := XiINV (req[0], reqBarJyCLB0);
  InvReq2JyCLB := XiINV (req[2], reqBarJyCLB2);

  AoJy := AO (jyTerm[0], req[3], jyTerm[1], req[1], jy);

  OrTermJy[0] := XiOR2(x, reqBarJyCLB2, jyTerm[0]);
  OrTermJy[1] := XiOR2(xBarJyCLB, reqBarJyCLB0, jyTerm[1]);


  (* Ky *)

  ClbKy := XiCLBMAP5i1o (x, req[0..3], ky);
  InvXKyCLB := XiINV (x, xBarKyCLB);
  InvReq1KyCLB := XiINV (req[1], reqBarKyCLB1);
  InvReq3KyCLB := XiINV (req[3], reqBarKyCLB3);

  AoKy := AO (kyTerm[0], req[2], kyTerm[1], req[0], ky);

  OrTermKy[0] := XiOR2(xBarKyCLB, reqBarKyCLB1, kyTerm[0]);
  OrTermKy[1] := XiOR2(x, reqBarKyCLB3, kyTerm[1]);


  ClbJxKx := XiCLBMAP5i2o(req[0], req[1], req[2], y, req[3], jx, kx);

  InvReqJxKxCLB1 := XiINV (req[1], reqBarJxKxCLB1);
  InvReqJxKxCLB3 := XiINV (req[3], reqBarJxKxCLB3);

  (* Jx  *)

  AndJx := XiAND2(jxFact[0..1], jx);
  OrFactJx[0] := XiOR2(y, reqBarJxKxCLB1, jxFact[0]);
  OrFactJx[1] := XiOR2(req[3], req[2], jxFact[1]);

  (* Kx  *)

  AndKx := XiAND2(kxFact[0..1], kx);
  OrFactKx[0] := XiOR2(y, reqBarJxKxCLB3, kxFact[0]);
  OrJactKx[1] := XiOR2(req[0], req[1], kxFact[1]);

  AnyOr := XiOR4(req[0..3], anyReq);
  AndKout := XiAND2 (anyReq, routeEnable, kOut);
  FFOutDis := JKFF (fs, kOut, clock, outputDisable, outputEnable);
END;
```

3.38.3 The Correctness Statement

The proofs for the `grant` and `outputDisable` signals are treated separately. For both, active and inactive frames are then treated separately, and for an active frame the periods before and after the active time are considered separately. The correctness of the `outputDisable` signal follows immediately from the specifications for `OUTDIS`.

The grant cases for an inactive frame and active frame prior to the active time are the same. The grant signal should remain unchanged under the absence of an active signal: that is while routeEnable is low and there is no request on ltReq. It follows immediately from the specification of ARBITER_XY that if routeEnable is low, both bits of the grant signal are unchanged. If none of the bits of ltReq are high grant is also unchanged. The values of its two bits depend on the definitions ArbX and ArbY respectively. When all the bits of ltReq are low the definition ArbX becomes:

```
ArbX ltReq routeEnable y x =
      (routeEnable
       ⇒ (x ⇒ ( y ∧ F ∧ T ∧ T)
            |  ((y ∨ T) ∧(F ∨ F)))
       |  x)
```

This is equivalent to:

```
ArbX ltReq routeEnable y x =
      (routeEnable
       ⇒ (x ⇒ T |  F)
       |  x)
```

or

```
ArbX ltReq routeEnable y x =
      (routeEnable ⇒ x |  x)
```

Thus, ArbX has the value x and the corresponding bit of grant is unchanged. A similar argument follows with the definition of ArbY and the other bit of grant.

For the grant signal in the last part of an active frame, we must show that the correct arbitration decision is made causing grant to have the appropriate value. We must also show that this value is held stable until the end of the frame. The second part follows from the fact that routeEnable is low until the end of the cycle. The first part involves showing that GrantForOut returns the same result as ArbX and ArbY when routeEnable is true. This involves fiddly, but otherwise straightforward reasoning about the definition of round robin arbitration.

```
⊢ FOR ltReq ::(PSIGLEN 4).
     FOR grant ::(PSIGLEN 2).
       ARBITER ((ltReq,routeEnable,frameStart),grant,outputDisable) ⊃
       ARBITER_SPEC ((ltReq,routeEnable,frameStart),grant,outputDisable)
```

3.39 ARBITERS

3.39.1 The Behavioural Specification

ARBITERS performs round-robin arbitration for all the output ports. It is equivalent to a series of individual arbiters, one per output port. Each input and output (other than routeEnable and frameStart which are single bits) consists of a word holding values of the type required for a single arbiter.

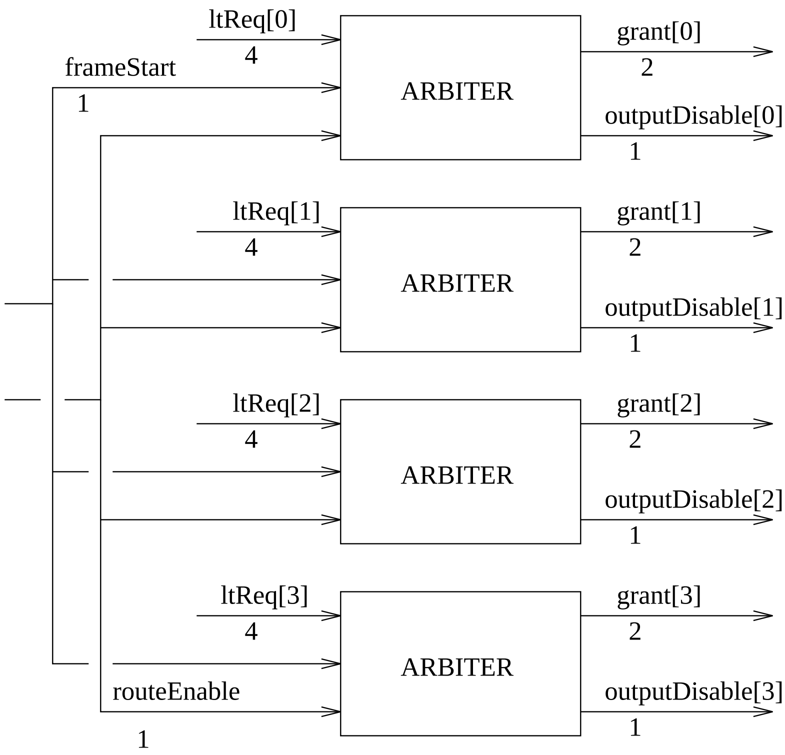

Figure 29: The Implementation of `ARBITERS`

```
⊢ ∀ ltReq routeEnable frameStart grant outputDisable.
     ARBITERS_SPEC ((ltReq,routeEnable,frameStart),grant,outputDisable) =
     (FOR i ::(TO (SIGLEN grant)).
       ARBITER_SPEC
         ((SBITS i ltReq,routeEnable,frameStart),
          SBIT i grant,
          SBIT i outputDisable))
```

3.39.2 The Structural Specification

`ARBITERS` provides an extra layer of hierarchy. It combines the separate arbiters into a single arbitration unit.

```
⊢ ∀ ltReq routeEnable frameStart grant outputDisable.
     ARBITERS ((ltReq,routeEnable,frameStart),grant,outputDisable) =
     (FOR i ::(TO (SIGLEN grant)).
       ARBITER
         ((SBITS i ltReq,routeEnable,frameStart),
          SBIT i grant,
          SBIT i outputDisable))
```

```
⊢ ∀ ltReq routeEnable frameStart grant outputDisable.
     ARBITERS_SIMPL ((ltReq,routeEnable,frameStart),grant,outputDisable) =
     (FOR i ::(TO (SIGLEN grant)).
       ARBITER_SPEC
         ((SBITS i ltReq,routeEnable,frameStart),
          SBIT i grant,
          SBIT i outputDisable))
```

Qudos HDL

```
Arb[0-3] := ARBITER (ltReq[0-3], ltReq[4-7], ltReq[8-11], ltReq[12-15],
                     clock,
                     routeEnable, frameStart, xGrant[0-3], yGrant[0-3],
                     outputDisable[0-3]);
```

3.39.3 The Correctness Statement

The correctness theorem follows immediately from the behavioural and structural definitions and the correctness theorem for `ARBITERS`.

```
⊢ FOR ltReq ::(PSIG2LEN 4 4).
     FOR grant ::(PSIG2LEN 4 2).
       ARBITERS ((ltReq,routeEnable,frameStart),grant,outputDisable) ⊃
       ARBITERS_SPEC ((ltReq,routeEnable,frameStart),grant,outputDisable)
```

3.40 PRIORITY_DECODE

3.40.1 The Behavioural Specification

`PRIORITY_DECODE` decodes the headers for all the output ports. It takes as inputs a word of active bits, `active`, a word of priority bits, `priority`, and a word of 2-bit requests `route`, all from the headers. Its output, `ltReq`, is a word of position vectors indicating which inputs are making successful requests for each output (after priority decoding but before arbitration). `ltReq` has one position vector for each input port. Each position vector has one bit corresponding to each output port. If it is true then the input port is making a request for the output port.

Each bit in `ltReq` corresponding to an input port–output port pair is set as follows. If there is a high priority request from any input port for the output port and there is a high priority request from the input port for it then the bit is set. If there is no high priority request for the output port then the bit is set if there is any request from the input port for it. `IsHiReq` and `IsGenReq` indicate if a high or general request are being made, respectively.

A general priority request is being made if the active bit is set for the input port, and its route field when converted to a number is the number of the output port under consideration. A high priority request is being made if in addition the priority bit is set.

Priority decoding occurs on all bytes arriving from the input ports. However, the results will be ignored by the arbiter except when they correspond to headers. The timing circuitry controls this.

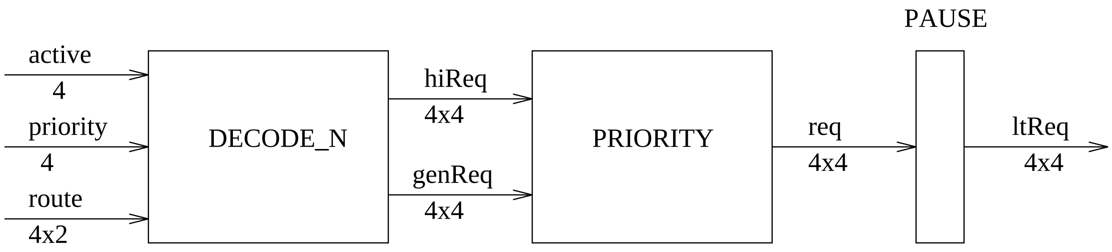

Figure 30: The Implementation of PRIORITY_DECODE

```
⊢ ∀ inport outport active priority route.
    IsHiReq inport outport active priority route =
    BIT inport active ∧
    BIT inport priority ∧
    (BNVAL (BIT inport route) = outport)
```

```
⊢ ∀ inport outport active route.
    IsGenReq inport outport active route =
    BIT inport active ∧ (BNVAL (BIT inport route) = outport)
```

```
⊢ ∀ active priority route ltReq.
    PRIORITY_DECODE_SPEC ((active,priority,route),ltReq) =
    (∀ t.
      ltReq (t + 1) =
      MKW 4
        (λ i.
          MKW 4
            (λ n.
              (∃ k. k < 4 ∧ IsHiReq k n (active t) (priority t) (route t))
              ⇒ (IsHiReq i n (active t) (priority t) (route t))
              |  (IsGenReq i n (active t) (route t)))))
```

3.40.2 The Structural Specification

PRIORITY_DECODE gives an extra level of hierarchy over the HDL version. It contains the hardware that is used to decode the headers. PAUSE is used for the latch.

```
⊢ ∀ active priority route ltReq.
    PRIORITY_DECODE ((active,priority,route),ltReq) =
    (LOCAL hiReq genReq req ::(PSIG2LEN 4 4).
      DECODE_N ((active,priority,route),hiReq,genReq) ∧
      PRIORITY ((hiReq,genReq),req) ∧
      PAUSE (req,ltReq))
```

```
⊢ ∀ active priority route ltReq.
     PRIORITY_DECODE_SIMPL ((active,priority,route),ltReq) =
     (LOCAL hiReq genReq req ::(PSIG2LEN 4 4).
       DECODE_N_SPEC ((active,priority,route),hiReq,genReq) ∧
       PRIORITY_SPEC ((hiReq,genReq),req) ∧
       PAUSE_SPEC (req,ltReq))
```

Qudos HDL

```
Decode[0] := DECODE (d[0..3],   hiReq[0..3],   genReq[0..3]);
Decode[1] := DECODE (d[8..11],  hiReq[4..7],   genReq[4..7]);
Decode[2] := DECODE (d[16..19], hiReq[8..11],  genReq[8..11]);
Decode[3] := DECODE (d[24..27], hiReq[12..15], genReq[12..15]);


PriFilter[0-3] := PRIFIL4CLB (
                hiReq[0-3], hiReq[4-7], hiReq[8-11], hiReq[12-15],
                genReq[0-3], genReq[4-7], genReq [8-11], genReq[12-15],
                req[0-3], req[4-7], req[8-11], req[12-15]);

FFReq[0-15] := XiDFFd(req[0-15], clock, ltReq[0-15]);
```

3.40.3 The Correctness Statement

```
⊢ FOR active priority ::(PSIGLEN 4).
     FOR route ::(PSIG2LEN 4 2).
       FOR ltReq ::(PSIG2LEN 4 4).
         PRIORITY_DECODE ((active,priority,route),ltReq) ⊃
         PRIORITY_DECODE_SPEC ((active,priority,route),ltReq)
```

3.41 ARBITRATION

3.41.1 The Behavioural Specification

ARBITRATION is the top-level specification of the full decoding, prioritising arbitration unit of the fabric. Two signals are output: grant which indicates which input port is being granted access to the input port at a given time, and outputDisable which indicates whether that value is currently valid or should be ignored. grant only changes once per frame, just after the new cells arrive. outputDisable is held high from the start of the cycle until a new decision has been made just after the cells arrive.

```
⊢ ∀ act pri req frameStart grant outputDisable.
    ARBITRATION_SPEC ((act,pri,req,frameStart),grant,outputDisable) =
    (∀ ts ta te.
      (AFRAME2 ts ta te frameStart act ⊃
       GRANT ts ta te act pri req grant ∧
       STABLE (ts + 1) (ta + 2) outputDisable
         (MKW (SIGLEN outputDisable) (λ i. T)) ∧
       STABLE (ta + 2) (te + 1) outputDisable
         (MKW (SIGLEN outputDisable)
           (λ i.
             Disable (act ta) (pri ta) (WMAP ($= i o BNVAL) (req ta))
               (BIT i (grant (ta + 1)))))) ∧
      (IFRAME ts te frameStart act ⊃
       STABLE (ts + 2) (te + 2) grant (grant (ts + 1)) ∧
       STABLE (ts + 1) (te + 1) outputDisable
         (MKW (SIGLEN outputDisable) (λ i. T))))
```

`GRANT` defines the values of the `grant` word of the arbitration unit over the period of a frame. Each output port grants some input port access at each point in time, and it is only changed (on a round robin basis) at a fixed time after the headers arrive.

```
⊢ ∀ ts ta te act pri req grant.
    GRANT ts ta te act pri req grant =
    STABLE (ts + 2) (ta + 2) grant (grant (ts + 1)) ∧
    STABLE (ta + 2) (te + 2) grant
      (MKW (SIGLEN grant)
        (λ i.
          Grant (act ta) (pri ta) (WMAP ($= i o BNVAL) (req ta))
            (BIT i (grant (ta + 1)))))
```

`Grant` defines the value of the `grant` signal for a single output port given the active statuses, priority statuses and requests made by each of the inport ports and the grant made by this output port previously.

If no input port is requesting the output port, then the input port last granted access to this output port continues to be granted it. Otherwise the new round robin arbitrated port is granted access to it.

```
⊢ ∀ act pri req last.
    Grant act pri req last =
    (let suc_inport = PickSuccessfulInput act pri req (BNVAL last)
     in
     ((suc_inport = NO_RESULT) ⇒ last | (NBWORD 2 (ResultOf suc_inport))))
```

`DISABLE` defines the values of the `outputDisable` word of the arbitration unit over the period of a frame. Outputs are disabled until a given point after the cells arrive, after which it takes on a new value dependent on the input values at the cell arrival time. Outputs are then disabled if no request for this output port was made in the current frame. Otherwise they are enabled.

```
⊢ ∀ ts ta te act pri req grant outputDisable.
     DISABLE ts ta te act pri req grant outputDisable =
     ((ta < te)
      ⇒ (STABLE (ts + 1) (ta + 2) outputDisable
           (MKW (SIGLEN outputDisable) (λ i. T)))
      | (STABLE (ts + 1) (ta + 1) outputDisable
          (MKW (SIGLEN outputDisable) (λ i. T)))) ∧
     STABLE (ta + 2) (te + 1) outputDisable
       (MKW (SIGLEN outputDisable)
         (λ i.
           Disable (act ta) (pri ta) (WMAP ($= i o BNVAL) (req ta))
             (BIT i (grant (ta + 1)))))
```

`Disable` defines the value of the `outputDisable` signal for a single output port given the active statuses, priority statuses and requests made by each of the inport ports and the grant made by this output port previously.

Outputs are disabled if no request for this output port was made in the current frame.

```
⊢ ∀ act pri req last.
     Disable act pri req last =
     (let suc_inport = PickSuccessfulInput act pri req (BNVAL last)
      in
      (suc_inport = NO_RESULT))
```

3.41.2 The Structural Specification

`ARBITRATION` gives an extra level of hierarchy over the HDL version. It contains all the hardware that is used to decode the headers. `PAUSE` is used for the latch.

```
⊢ ∀ active priority route frameStart grant outputDisable.
     ARBITRATION ((active,priority,route,frameStart),grant,outputDisable) =
     (LOCAL routeEnable.
       LOCAL ltReq ::(PSIG2LEN 4 4).
         PRIORITY_DECODE ((active,priority,route),ltReq) ∧
         TIMING ((frameStart,active),routeEnable) ∧
         ARBITERS ((ltReq,routeEnable,frameStart),grant,outputDisable))
```

```
⊢ ∀ active priority route frameStart grant outputDisable.
     ARBITRATION_SIMPL
       ((active,priority,route,frameStart),grant,outputDisable) =
     (LOCAL routeEnable.
       LOCAL ltReq ::(PSIG2LEN 4 4).
         PRIORITY_DECODE_SPEC ((active,priority,route),ltReq) ∧
         TIMING_SPEC ((frameStart,active),routeEnable) ∧
         ARBITERS_SPEC ((ltReq,routeEnable,frameStart),grant,outputDisable))
```

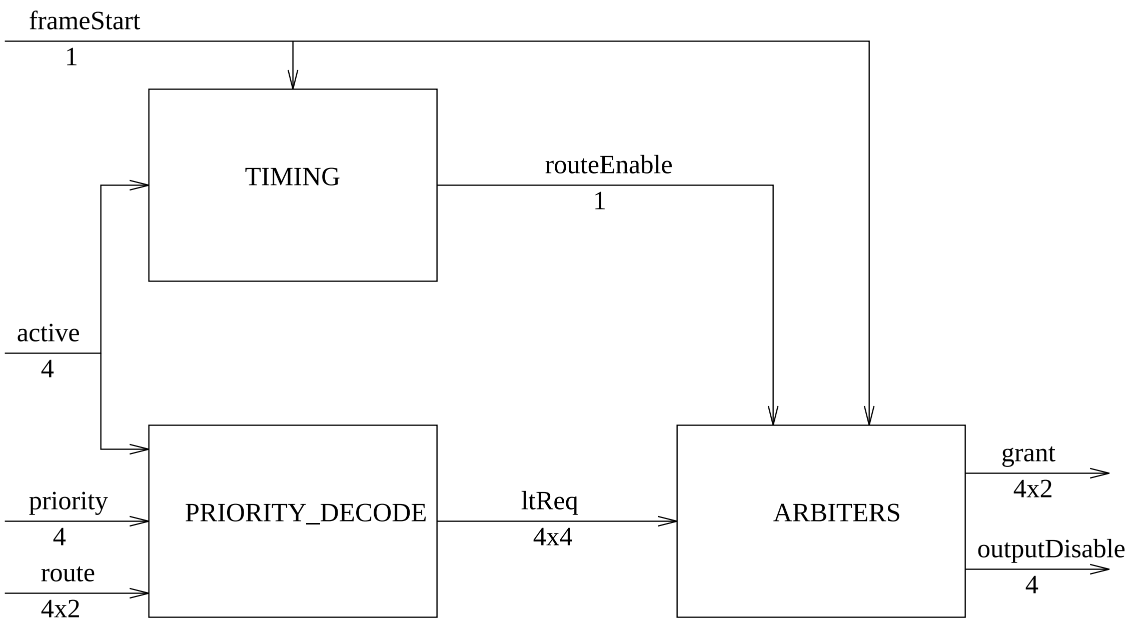

Figure 31: The Implementation of ARBITRATION

Qudos HDL

```
Timing := TIMING (frameStart, clock, d[0], d[8], d[16], d[24],
                  routeEnable);


Decode[0] := DECODE (d[0..3],   hiReq[0..3],   genReq[0..3]);
Decode[1] := DECODE (d[8..11],  hiReq[4..7],   genReq[4..7]);
Decode[2] := DECODE (d[16..19], hiReq[8..11],  genReq[8..11]);
Decode[3] := DECODE (d[24..27], hiReq[12..15], genReq[12..15]);


PriFilter[0-3] := PRIFIL4CLB (
                hiReq[0-3], hiReq[4-7], hiReq[8-11], hiReq[12-15],
                genReq[0-3], genReq[4-7], genReq [8-11], genReq[12-15],
                req[0-3], req[4-7], req[8-11], req[12-15]);

FFReq[0-15] := XiDFFd(req[0-15], clock, ltReq[0-15]);

Arb[0-3] := ARBITER (ltReq[0-3], ltReq[4-7], ltReq[8-11], ltReq[12-15],
                     clock,
                     routeEnable, frameStart, xGrant[0-3], yGrant[0-3],
                     outputDisable[0-3]);
```

3.41.3 The Correctness Statement

As with the earlier modules, the proof is split into cases on the frame type and the intervals within the frame. We must show that the `grant` and `outputDisable` signals have the values as specified by `ARBITRATION_SPEC`.

`ARBITERS` uses a different notion of active and inactive frames to that used by `ARBITRATION`. In the former, the active signal depends on the `ltReq` and `routeEnable` signals, whereas in the latter it depends on the active bit of the data line. Also the conditions on the bounds differ. We prove that if `ts` and `te` are the bounds of an inactive frame as used by `ARBITRATION` then they are also the bounds of an inactive frame as used by `ARBITERS`, assuming that `ltReq` is as specified by `PRIORITY_DECODE_SPEC`. Thus when `ARBITRATION` sees an inactive frame, so does `ARBITERS`. Similarly, if `ARBITRATION` sees an active frame, then each individual arbiter within `ARBITERS` sees either an active frame or an inactive frame, assuming that `routeEnable` is as specified by `TIMING`. An arbiter within `ARBITERS` can see an inactive frame even though `ARBITRATION` sees an active frame when there are requests within the frame for outputs other than that which the individual arbiter is responsible.

When proving the cases of the correctness theorem for `ARBITRATION`, we have the assumption that either an active or inactive `ARBITRATION` frame occurs. We can therefore deduce that an `ARBITER` active or inactive frame occurs and combine this with the definition `ARBITER_SPEC` to give values on the `grant` and `outputDisable` lines over the various intervals. These are in terms of the values of `ltReq` and `routeEnable`. We use the specification of `PRIORITY_DECODE` to convert them to being in terms of the active bit of the data line. The resulting expressions can then be simplified to the form specified by `ARBITRATION_SPEC`. This is done using lemmas about the underlying definitions of round robin arbitration and priority decoding.

```
⊢ FOR active priority outputDisable ::(PSIGLEN 4).
    FOR grant route ::(PSIG2LEN 4 2).
      ~ (frameStart 0) ∧
      (∀ t. ~ (frameStart (t + 1)) ∨ ~ (EXISTSABIT I (active t))) ⊃
      ARBITRATION ((active,priority,route,frameStart),grant,outputDisable) ⊃
      ARBITRATION_SPEC
        ((active,priority,route,frameStart),grant,outputDisable)
```

3.42 FAB4B4

3.42.1 The Behavioural Specification

`FAB4B4` provides the functionality of a switching element but has no input or output latches or buffers, thus its timing is slightly different to that of the full element. The behavioural specification is defined in terms of some auxiliary functions that we describe first.

`DEFAULT_DATA` specifies the data that is output to a port when it has not been requested by an input.

```
⊢ DEFAULT_DATA = ZEROW 8
```

`FABRIC4x4_ACK2` specifies the acknowledgement signal to be sent to each input port between times `ts` and `te`, with the headers arriving at time `ta`. It is given the data bytes input from each port `data_in`, the previous grants made `grant` and the acknowledgements from each output port `ack_ins`. It specifies the value of `ack_out` within the frame. Round robin arbitration is performed for each output. From the start of the cycle until a new decision is made, all inputs are sent negative acknowledgements. If the output port that an inputs header is requesting chooses that input, then the acknowledgement from the requested output port is sent to the input port for the remainder of the frame. Otherwise a negative acknowledgement is sent.

```
⊢ ∀ data_in fs last ack_in ack_out.
     FABRIC4x4_ACK2 (data_in,fs,last,ack_in,ack_out) =
     (∀ ts te ta.
       (IFRAME ts te fs (Actives o data_in) ⊃
        STABLE (ts + 1) (te + 1) ack_out (ZEROW (SIGLEN ack_out))) ∧
       (AFRAME2 ts ta te fs (Actives o data_in) ⊃
        STABLE (ts + 1) (ta + 2) ack_out (ZEROW (SIGLEN ack_out)) ∧
        DURING (ta + 2) (te + 1) ack_out
          (λ t.
            MKW (SIGLEN ack_out)
              (λ i.
                Fabric4x1Ack (data_in ta) (WMAP BNVAL (last (ta + 1)))
                  (ack_in t)
                  i))))
```

`FABRIC4x4_DATA_OUT2` specifies the data to be output to each output port between times `ts` and `te`, with the headers arriving at time `ta`. It is given the data bytes input from each port `data_in` and the previous grants made, `last`. It specifies the value of `data_out` within the frame. Round robin arbitration is performed for each output. From the start of the cycle until a new decision is made, all inputs are sent the default data `default_data`. After a delay, whilst a decision is made, the data from the successful inputs is sent to the appropriate output ports. The header bytes are not sent. Outputs for which there are no requests continue to receive default data.

```
⊢ ∀ default_data_out data_in fs last data_out.
     FABRIC4x4_DATA_OUT2 default_data_out (data_in,fs,last,data_out) =
     (∀ ts te ta.
       (IFRAME ts te fs (Actives o data_in) ⊃
        STABLE (ts + 2) (te + 2) data_out
          (MKCW (SIGLEN data_out) default_data_out)) ∧
       (AFRAME2 ts ta te fs (Actives o data_in) ⊃
        STABLE (ts + 2) (ta + 3) data_out
          (MKCW (SIGLEN data_out) default_data_out) ∧
        DURING (ta + 3) (te + 2) data_out
          (λ t.
            MKW (SIGLEN data_out)
              (λ i.
                Fabric4x1DataOut default_data_out (data_in ta)
                  (data_in (t - 2))
                  (BIT i (WMAP BNVAL (last (ta + 1))))
                  i))))
```

`FABRIC4x4_LAST2` specifies the input ports most recently granted access to output ports between times `ts` and `te`, with the headers arriving at time `ta`. It is given the data bytes input from each port `data_in`. It specifies the value of `last` within the frame. Round robin arbitration is performed for each output. Except at a single point within the interval when a new decision is made, the value remains constant. At the time when a new decision is made, the value for each output port is updated.

```
⊢ ∀ d fs last.
     FABRIC4x4_LAST2 (d,fs,last) =
     (∀ ts te ta.
       (IFRAME ts te fs (Actives o d) ⊃
        STABLE (ts + 2) (te + 2) last (last (ts + 1))) ∧
       (AFRAME2 ts ta te fs (Actives o d) ⊃
        STABLE (ts + 2) (ta + 2) last (last (ts + 1)) ∧
        STABLE (ta + 2) (te + 2) last
          (MKW (SIGLEN last)
            (λ i.
              Grant (Actives (d ta)) (Priorities (d ta))
                (WMAP ($= i) (Requests (d ta)))
                (BIT i (last (ta + 1)))))))
```

`FAB4B4_SPEC` gives the specification of the internals of the fabric element. That is without any of the external latches and gates. `FAB4B4` has periodic behaviour over a frame, determined by the `frameStart` signal. At a time `ta` within the frame, the headers arrive. This occurs at the first time after `ts` that any active bit in the data is high. Round robin arbitration is then performed using the information in the headers at that time. Until a decision is made, negative acknowledgements are sent to all inputs, and the default data is sent to all outputs. Thereafter the successful inputs are sent positive acknowledgements and their data (less the header) is sent to the outputs. The state recording the last grants made for each output port is updated.

```
⊢ ∀ default_data_out last data_in frame_start ack_in data_out ack_out.
     FAB4B4_SPEC default_data_out last
       ((data_in,frame_start,ack_in),data_out,ack_out) =
     FABRIC4x4_DATA_OUT2 default_data_out
       (data_in,frame_start,last,data_out) ∧
     FABRIC4x4_ACK2 (data_in,frame_start,last,ack_in,ack_out) ∧
     FABRIC4x4_LAST2 (data_in,frame_start,last)
```

3.42.2 The Structural Specification

`FAB4B4` is the definition of the implementation of the 4 by 4 switching element less the external input and output buffers and latches. It is constructed from three units: the acknowledgement unit, the dataswitch (which includes delay circuitry), and the arbitration unit.

On each frame, as indicated by the frameStart signal the cells from each input port are read in. Within each frame, 0s are read in from each port until the header bytes. The start of packet bit in each header ensures that an active header does contain a 1. The header bytes are read by the arbitration unit, where clashes are arbitrated. It outputs a grant signal for each output port indicating which input if any it is accepting and a corresponding set of disable signals which indicate when the grant signal is valid. These signals are read by the dataswitch which routes the data from the selected input ports to the appropriate output ports for the remainder of the cell. They are also read by the acknowledgement unit which sends negative acknowledgements to the inputs which were not selected, and routes the acknowledgement signal from the selected output port to the corresponding successful input port. This allows the output port to refuse to accept packets, even when there are no clashes within the element.

```
⊢ ∀ d frameStart ackIn dOut ackOut.
     FAB4B4 ((d,frameStart,ackIn),dOut,ackOut) =
     (LOCAL grant ::(PSIG2LEN 4 2).
       LOCAL outputDisable ::(PSIGLEN 4).
         ARBITRATION
           ((SBITS 0 d,SBITS 1 d,SWSEGS 2 2 d,frameStart),
            grant,
            outputDisable) ∧
         PAUSE_DATASWITCH ((d,grant,outputDisable),dOut) ∧
         ACK ((ackIn,grant,outputDisable),ackOut))
```

```
⊢ ∀ d frameStart ackIn dOut ackOut.
     FAB4B4_SIMPL ((d,frameStart,ackIn),dOut,ackOut) =
     (LOCAL grant ::(PSIG2LEN 4 2).
       LOCAL outputDisable ::(PSIGLEN 4).
         ARBITRATION_SPEC
           ((SBITS 0 d,SBITS 1 d,SWSEGS 2 2 d,frameStart),
            grant,
            outputDisable) ∧
         PAUSE_DATASWITCH_SPEC ((d,grant,outputDisable),dOut) ∧
         ACK_SPEC ((ackIn,grant,outputDisable),ackOut))
```

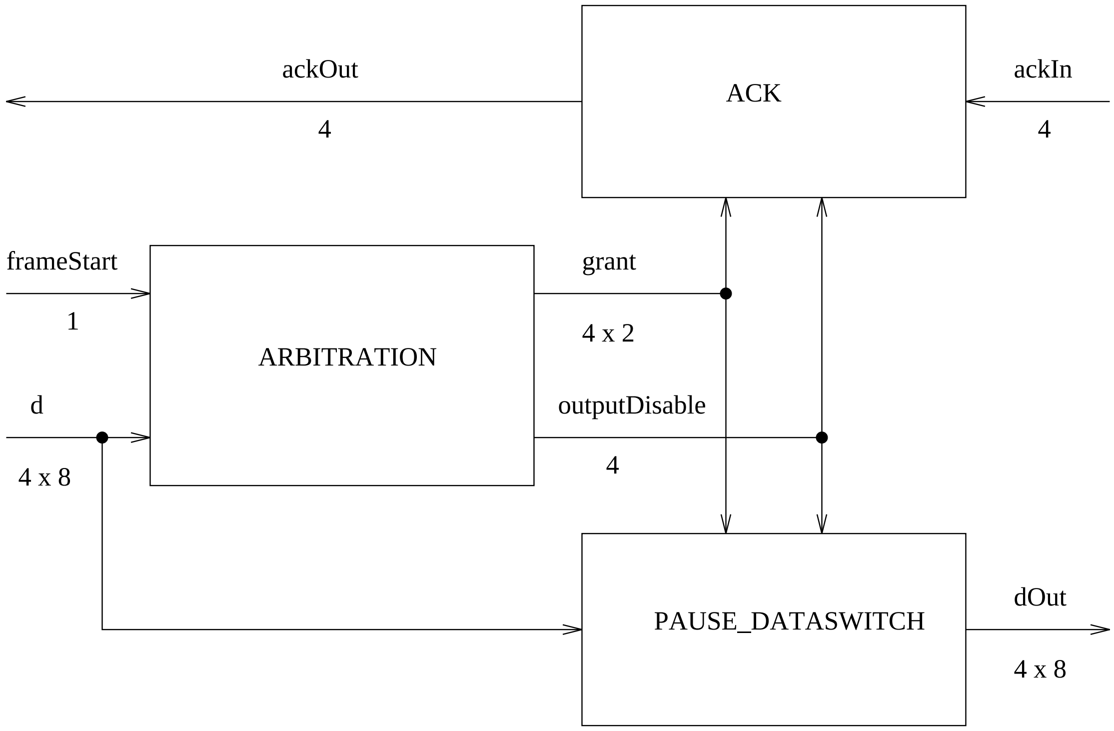

Figure 32: The Implementation of FAB4B4

Qudos HDL

```
    Timing := TIMING (frameStart, clock, d[0], d[8], d[16], d[24],
                      routeEnable);


    Decode[0] := DECODE (d[0..3],   hiReq[0..3],   genReq[0..3]);
    Decode[1] := DECODE (d[8..11],  hiReq[4..7],   genReq[4..7]);
    Decode[2] := DECODE (d[16..19], hiReq[8..11],  genReq[8..11]);
    Decode[3] := DECODE (d[24..27], hiReq[12..15], genReq[12..15]);


    PriFilter[0-3] := PRIFIL4CLB (
                hiReq[0-3], hiReq[4-7], hiReq[8-11], hiReq[12-15],
                genReq[0-3], genReq[4-7], genReq [8-11], genReq[12-15],
                req[0-3], req[4-7], req[8-11], req[12-15]);

    FFReq[0-15] := XiDFFd(req[0-15], clock, ltReq[0-15]);

    Arb[0-3] := ARBITER (ltReq[0-3], ltReq[4-7], ltReq[8-11], ltReq[12-15],
                         clock,
                         routeEnable, frameStart, xGrant[0-3], yGrant[0-3],
                         outputDisable[0-3]);

    Pause[0-31] := XiDFFd(d[0-31], clock, dPause[0-31]);

    DSw[0] := DATASWITCH (dPause[0..31], clock, xGrant[0], yGrant[0],
                                outputDisable[0], dOut[0..7]);
    DSw[1] := DATASWITCH (dPause[0..31], clock, xGrant[1], yGrant[1],
                                outputDisable[1], dOut[8..15]);
    DSw[2] := DATASWITCH (dPause[0..31], clock, xGrant[2], yGrant[2],
                                outputDisable[2], dOut[16..23]);
    DSw[3] := DATASWITCH (dPause[0..31], clock, xGrant[3], yGrant[3],
                                outputDisable[3], dOut[24..31]);

    AckGen[0-3] := ACKGEN (ackIn[0-3], xGrant[0-3], yGrant[0-3],
                   outputDisable[0-3],
                   ackTerm[0-3], ackTerm[4-7], ackTerm[8-11], ackTerm[12-15]);

    AckOr[0] := ACKOR (ackTerm[0..3], ackOut[0]);
    AckOr[1] := ACKOR (ackTerm[4..7], ackOut[1]);
    AckOr[2] := ACKOR (ackTerm[8..11], ackOut[2]);
    AckOr[3] := ACKOR (ackTerm[12..15], ackOut[3]);
END;
```

3.42.3 The Correctness Statement

We have proved that for all input and output signals for which the structural specification holds, there exists a signal `last` of the appropriate size for which the behavioural specification holds. That is, any behaviour exhibited by the structural specification is permitted by the behavioural specification. It is assumed that the frame start signal does not go high in the first clock cycle and that thereafter the headers do not arrive in a cycle prior to the next frame start signal going high.

If this condition does not hold, the `outputDisable` signal may not reset itself correctly.

```
⊢ FOR ackIn ackOut ::(PSIGLEN 4).
    FOR dOut d ::(PSIG2LEN 4 8).
      ~ (frameStart 0) ∧
      (∀ t. ~ (frameStart (t + 1)) ∨ ~ (EXISTSABIT I (SBITS 0 d t))) ⊃
      FAB4B4 ((d,frameStart,ackIn),dOut,ackOut) ⊃
      (LOCAL last ::(PSIG2LEN 4 2).
        FAB4B4_SPEC DEFAULT_DATA last ((d,frameStart,ackIn),dOut,ackOut))
```

Each of the signals `ackOut`, `dOut` and `last` are considered separately. For each the proof is split into cases on the type of frame and the intervals within a frame. The specification of `ARBITRATION` gives the values of `grant` and `outputDisable` in terms of the input signals to `FAB4B4`. There is no frame structure in the specifications of `ACK` and `PAUSE_DATASWITCH`. They just give the values of the outputs in terms of the values of their inputs on the previous cycles. These inputs include `grant` and `outputDisable`. They can be eliminated using the specification of `ARBITRATION` over the intervals it specifies. For example, for the signal `ackOut` we prove the theorem:

```
⊢ (SIGLEN outputDisable = SIGLEN ackOut) ∧
   ACK_SPEC ((ackIn,grant,outputDisable),ackOut) ∧
   STABLE (ts + 1) (ta + 2) outputDisable
     (MKW (SIGLEN outputDisable) (λ i. T)) ⊃
   STABLE (ts + 1) (ta + 2) ackOut (ZEROW (SIGLEN ackOut))
```

This gives the value of `ackOut` in the interval up to the active signal arriving using the information that `outputDisable` is high throughout this period.

3.43 The Switching Fabric—`FABRIC4x4`

In this section we describe the top level of the switching fabric design. We give behavioural and structural specifications. We then give the main correctness theorem we have proved which states that the implementation of the switching element satisfies the specified behaviour, under certain assumptions about the input signals. It should be noted that the correctness theorem talks about a description of the implementation rather than the implementation itself. The correctness theorem will say nothing about the implementation as actually fabricated, if this does not correspond to the description used.

3.43.1 The Behavioural Specification

The predicate `FABRIC4B4_SPEC` describes the behaviour of the 4 by 4 switching element. It takes three inputs. The first, `dataIn` is a word of 4 data bytes: one from each input port. Cells are injected into the fabric a byte at a time on this input. The second `frameStart` is the frame start signal which is used to synchronise the injection of cells from the different input ports. It controls the timing of when new routeing decisions are made. Finally, the `ackIn` signal carries one bit of acknowledgement information back from each output port, to indicate whether it is willing to accept a cell. The fabric has two outputs. The first, `dataOut` is a word of 4 data bytes: one for each output port. The cells are sent to the appropriate output port on `dataOut`. The second output, `ackIn`, carries one bit of acknowledgement information back to each input port. This indicates whether the cell clashed with other cells or was rejected by the requested output port. The fabric retains one word of state, `last`. This word consists of one entry for each output port indicating the last input port from which it received data. This information is used to arbitrate between cell clashes. The definition is also parameterised by the byte value sent to output ports, when they are not being sent a cell. The current implementation of the port controllers requires such bytes to have a zero in the active bit.

`FABRIC4B4` has periodic behaviour over a frame, determined by the `frameStart` signal. At a synchronised time within each frame, the headers arrive. The fabric recognises this time by watching the active bits in the data. When one or more go high the cells have arrived. Round robin arbitration is then performed using the information in the headers at that time. Until a decision is made, negative acknowledgements are sent to all inputs, and the default data is sent to all outputs. Thereafter the successful inputs are sent positive acknowledgements and their data (less the header) is sent to the outputs. The state recording the last grants made for each output port is updated.

The specification is split into three independent parts. `FABRIC4x4_DATA_OUT` specifies the behaviour of the `dataOut` signal. `FABRIC4x4_ACK` specifies the behaviour of the `ackOut` signal. `FABRIC4x4_LAST` specifies the state on each cycle (represented by the signal `last`).

```
⊢ ∀ defaultDataOut last dataIn frameStart ackIn dataOut ackOut.
     FABRIC4B4_SPEC defaultDataOut last
       ((dataIn,frameStart,ackIn),dataOut,ackOut) =
     FABRIC4x4_DATA_OUT defaultDataOut (dataIn,frameStart,last,dataOut) ∧
     FABRIC4x4_ACK (dataIn,frameStart,last,ackIn,ackOut) ∧
     FABRIC4x4_LAST (dataIn,frameStart,last)
```

`FABRIC4x4_ACK` specifies the acknowledgement signals to be sent to each input port. It takes as arguments the data signal from the input ports port, `dataIn`, the frame start signal, `fs`, the previous grants made, `last`, the acknowledgement signals from each output port, `ackIn`, and the acknowledgement signals to the input ports `ackOut`.

The specification is given with respect to frames. Given a pair of times `ts` and `te` which define the bounds of an inactive frame, from time `ts+1` to `te+1` negative acknowledgements will be output on all `ackOut` lines.

Given a triple of times `ts`, `ta` and `te` which define an active frame, from time `ts+1` to `ta+3` zeroes will be output on all `ackOut` lines. From then to time `te+1` the value output to an input port will depend on whether that input ports cell was successful and whether the successful output port is receiving cells. The value for each input port (ie each bit of `ackOut`) is specified independently by the function `Fabric4x1Ack`. It is passed the values of the headers from all input ports (the value of `dataIn` at time `ta`; the last chosen values for each output port identified; the acknowledgements from all the output ports and the number of the input port under consideration. Arbitration is performed on the requests given in the headers. If the request from the input port under consideration is successful it sees the acknowledgement signal from its chosen output port. Otherwise it continues to see a negative acknowledge.

```
⊢ ∀ dataIn fs last ackIn ackOut.
     FABRIC4x4_ACK (dataIn,fs,last,ackIn,ackOut) =
     (∀ ts te ta.
       (IFRAME ts te fs (Actives o dataIn) ⊃
        STABLE (ts + 1) (te + 1) ackOut (ZEROW (SIGLEN ackOut))) ∧
       (AFRAME ts ta te fs (Actives o dataIn) ⊃
        STABLE (ts + 1) (ta + 3) ackOut (ZEROW (SIGLEN ackOut)) ∧
        DURING (ta + 3) (te + 1) ackOut
          (λ t.
            MKW (SIGLEN ackOut)
              (λ i.
                Fabric4x1Ack (dataIn ta) (WMAP BNVAL (last (ta + 2))) (ackIn t)
                  i))))
```

FABRIC4x4_DATA_OUT specifies the data to be output to each output port over the period of a frame. It takes as arguments the data signal from the input ports port dataIn, the frame start signal fs, the previous grants made last and the outgoing data signal to the output ports, dataOut.

The specification is given with respect to frames. Given a pair of times ts and te which define the bounds of an inactive frame, from time ts+3 to te+3 the default data value will be output on dataOut to all output ports.

Given a triple of times ts, ta and te which define an active frame, from time ts+3 to ta+5 the default data value will be output on datOut to all output ports. From then to time te + 3 the value output to an output port will depend on which input port, if any, successfully requests the output port. The value for each output port is specified independently by the function Fabric4x1DataOut. If no input port requested the output port on this cycle, the default value is sent to the output port. Otherwise the data from the successful input port is output delayed by 4 cycles (dataIn (t-4)). Since this behaviour starts 5 cycles after the header arrived from the input port, the header is *not* forwarded. Arbitration between different input ports requesting the same output port is on a round robin basis. The most recent successful input for output i is held in bit i of the state signal last. It is accessed using BIT i (last (ta+2)) which specifies that the arbitration takes place 2 cycles after the header arrives. This is a binary representation of the port number. It is converted to a natural number using BNVAL.

```
⊢ ∀ defaultDataOut dataIn fs last dataOut.
     FABRIC4x4_DATA_OUT defaultDataOut (dataIn,fs,last,dataOut) =
     (∀ ts te ta.
       (IFRAME ts te fs (Actives o dataIn) ⊃
        STABLE (ts + 3) (te + 3) dataOut
          (MKCW (SIGLEN dataOut) defaultDataOut)) ∧
       (AFRAME ts ta te fs (Actives o dataIn) ⊃
        STABLE (ts + 3) (ta + 5) dataOut
          (MKCW (SIGLEN dataOut) defaultDataOut) ∧
        DURING (ta + 5) (te + 3) dataOut
          (λ t.
            MKW (SIGLEN dataOut)
              (λ i.
                Fabric4x1DataOut defaultDataOut (dataIn ta) (dataIn (t - 4))
                  (BIT i (WMAP BNVAL (last (ta + 2))))
                  i))))
```

FABRIC4x4_LAST specifies the input ports most recently granted access to each output ports. This information is held in the signal last. Each bit of last holds the information for one output port. It is dependent on the data signal from the input ports dataIn and the frame start signal fs which determines the timing.

The specification is given with respect to frames. Given a pair of times ts and te which define the bounds of an inactive frame, from time ts+2 to te+2 the value of last is unchanged from that at the end of the previous cycle.

On an active frame round robin arbitration is performed independently for each output. Except at a single point within the interval when a new decision is made, the value remains constant. At the time when a new decision is made, the value for each output port is updated. Given a triple of times ts, ta and te which define an active frame, from time ts+2 to ta+3, last does not change. By time ta+3 a new decision has been made and so from this time to te+2 last holds the new value. The arbitration result is specified by Grant. It depends on the active and priority bits of each of the headers, an indication of which of the headers are making requests for the output port under consideration, (for output port i we need to know if the request fields of each of the headers is equal to i), and the last successful input for this output (bit i of last at time ta+2).

```
⊢ ∀ dataIn fs last.
     FABRIC4x4_LAST (dataIn,fs,last) =
     (∀ ts te ta.
       (IFRAME ts te fs (Actives o dataIn) ⊃
        STABLE (ts + 2) (te + 2) last (last (ts + 1))) ∧
       (AFRAME ts ta te fs (Actives o dataIn) ⊃
        STABLE (ts + 2) (ta + 3) last (last (ts + 1)) ∧
        STABLE (ta + 3) (te + 2) last
          (MKW (SIGLEN last)
            (λ i.
              Grant (Actives (dataIn ta)) (Priorities (dataIn ta))
                (WMAP ($= i) (Requests (dataIn ta)))
                (BIT i (last (ta + 2)))))))
```

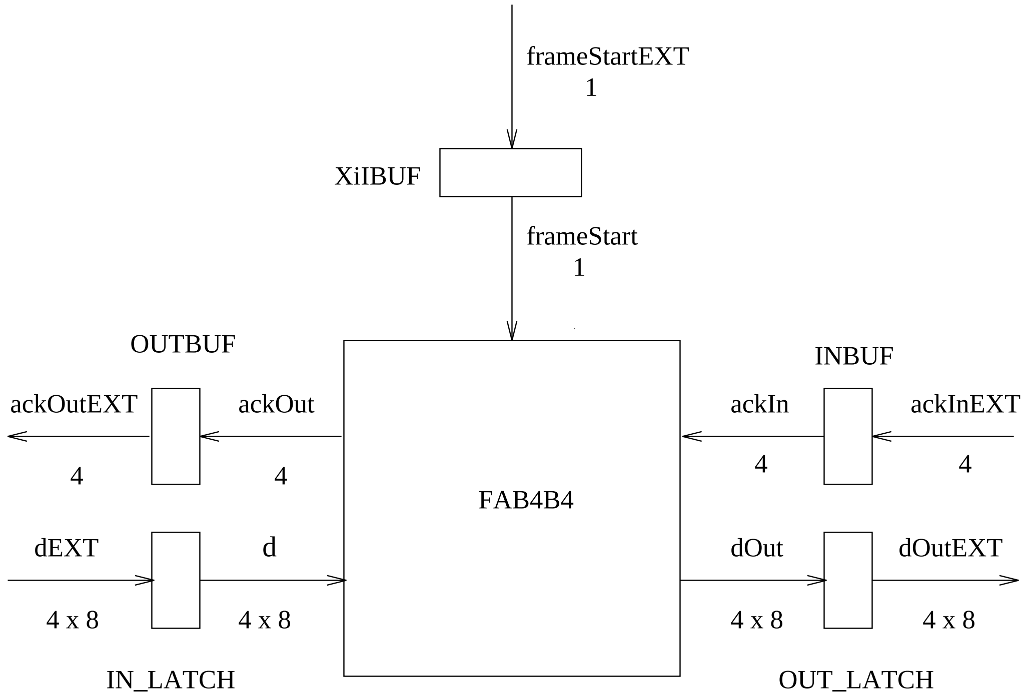

Figure 33: The Implementation of `FABRIC4B4`

3.43.2 The Structural Specification

`FABRIC4B4` is the definition of the implementation of the 4 by 4 switching element. The original HDL specification had little structure. For the HOL version we added extra layers of hierarchy to make the proof more tractable. This did not change the underlying design, only the description of it. The fabric consists of a main unit `FAB4B4` whose inputs and outputs are connected to input and output buffers or latches. Two versions of the HOL specification are provided: one in terms of the implementations of the components and one in terms of their specifications.

```
⊢ ∀ dEXT frameStartEXT ackInEXT dOutEXT ackOutEXT.
     FABRIC4B4 ((dEXT,frameStartEXT,ackInEXT),dOutEXT,ackOutEXT) =
     (LOCAL frameStart.
       LOCAL ackIn ackOut ::(PSIGLEN 4).
         LOCAL d dOut ::(PSIG2LEN 4 8).
           XiIBUF (frameStartEXT,frameStart) ∧
           IN_BUF (ackInEXT,ackIn) ∧
           IN_LATCH (dEXT,d) ∧
           OUT_LATCH (dOut,dOutEXT) ∧
           OUT_BUF (ackOut,ackOutEXT) ∧
           FAB4B4 ((d,frameStart,ackIn),dOut,ackOut))
```

```
⊢ ∀ default_data last dEXT frameStartEXT ackInEXT dOutEXT ackOutEXT.
     FABRIC4B4_SIMPL default_data last
       ((dEXT,frameStartEXT,ackInEXT),dOutEXT,ackOutEXT) =
     (LOCAL frameStart.
       LOCAL ackIn ackOut ::(PSIGLEN 4).
         LOCAL d dOut ::(PSIG2LEN 4 8).
           XiIBUF (frameStartEXT,frameStart) ∧
           IN_BUF_SPEC (ackInEXT,ackIn) ∧
           IN_LATCH_SPEC (dEXT,d) ∧
           OUT_LATCH_SPEC (dOut,dOutEXT) ∧
           OUT_BUF_SPEC (ackOut,ackOutEXT) ∧
           FAB4B4_SPEC default_data last ((d,frameStart,ackIn),dOut,ackOut))
```

Qudos HDL

```
DEF FABRIC4B4 (dEXT[0..31], clockEXT, frameStartEXT, ackInEXT[0..3]: IN;
               dOutEXT[0..31], ackOutEXT[0..3]: IO);

d[0..31], clock, frameStart, ackIn[0..3], ackOut[0..3],
dOut[0..31], dPause[0..31],

routeEnable, outputDisable[0..3],
hiReq[0..15], genReq[0..15], req[0..15],
ltReq[0..15],
xGrant[0..3], yGrant[0..3],
ackTerm[0..15]: IO;

BEGIN
    Tclk := XiGCLK (clockEXT, clock);
    FS := XiIBUF (frameStartEXT, frameStart);
    Ai[0-3] := XiIBUF (ackInEXT[0-3], ackIn[0-3]);
    I[0-31] := XiINFFd (dEXT[0-31], clock, d[0-31]);

    O[0-31] := XiOUTFFd (dOut[0-31], clock, dOutEXT[0-31]);
    Ao[0-3] := XiOBUF (ackOut[0-3], ackOutEXT[0-3]);


    Timing := TIMING (frameStart, clock, d[0], d[8], d[16], d[24],
                      routeEnable);


    Decode[0] := DECODE (d[0..3],   hiReq[0..3],   genReq[0..3]);
    Decode[1] := DECODE (d[8..11],  hiReq[4..7],   genReq[4..7]);
    Decode[2] := DECODE (d[16..19], hiReq[8..11],  genReq[8..11]);
    Decode[3] := DECODE (d[24..27], hiReq[12..15], genReq[12..15]);


    PriFilter[0-3] := PRIFIL4CLB (
                hiReq[0-3], hiReq[4-7], hiReq[8-11], hiReq[12-15],
                genReq[0-3], genReq[4-7], genReq [8-11], genReq[12-15],
                req[0-3], req[4-7], req[8-11], req[12-15]);

    FFReq[0-15] := XiDFFd(req[0-15], clock, ltReq[0-15]);
```

```
    Arb[0-3] := ARBITER (ltReq[0-3], ltReq[4-7], ltReq[8-11], ltReq[12-15],
                         clock,
                         routeEnable, frameStart, xGrant[0-3], yGrant[0-3],
                         outputDisable[0-3]);

    Pause[0-31] := XiDFFd(d[0-31], clock, dPause[0-31]);

    DSw[0] := DATASWITCH (dPause[0..31], clock, xGrant[0], yGrant[0],
                                outputDisable[0], dOut[0..7]);
    DSw[1] := DATASWITCH (dPause[0..31], clock, xGrant[1], yGrant[1],
                                outputDisable[1], dOut[8..15]);
    DSw[2] := DATASWITCH (dPause[0..31], clock, xGrant[2], yGrant[2],
                                outputDisable[2], dOut[16..23]);
    DSw[3] := DATASWITCH (dPause[0..31], clock, xGrant[3], yGrant[3],
                                outputDisable[3], dOut[24..31]);

    AckGen[0-3] := ACKGEN (ackIn[0-3], xGrant[0-3], yGrant[0-3],
                   outputDisable[0-3],
                   ackTerm[0-3], ackTerm[4-7], ackTerm[8-11], ackTerm[12-15]);

    AckOr[0] := ACKOR (ackTerm[0..3], ackOut[0]);
    AckOr[1] := ACKOR (ackTerm[4..7], ackOut[1]);
    AckOr[2] := ACKOR (ackTerm[8..11], ackOut[2]);
    AckOr[3] := ACKOR (ackTerm[12..15], ackOut[3]);
END;
```

3.43.3 The Correctness Statement

We have proved that for all input and output signals for which the structural specification holds, there exists a signal `last` of the appropriate size for which the behavioural specification holds. That is, any behaviour exhibited by the structural specification is permitted by the behavioural specification. It is assumed that the frame start signal does not go high in the first 2 clock cycles and that thereafter the headers do not arrive within 2 cycles prior to the next frame start signal. These conditions are sufficient for the implementation to have the correct behaviour. They ensure that the logic resets itself properly at the start of a cycle. The frame start signal should not be high just after headers arrive because if so the `TIMING` unit will ignore the headers. It should not go high two cycles after the headers arrive, because then the `outputDisable` signal may not reset itself correctly. The latter condition corresponds to the explicit assumption in the correctness statement for `FAB4B4`. The former is implicit in the definition of a frame used for `FAB4B4`. We have only proved that the conditions are sufficient, not that they are necessary. Other less restrictive conditions may also be sufficient to ensure correct behaviour. Indeed the assumption that we use states that the active bit should not be set in the last two cycles of *any* frame even if it was not set as part of a header. We believe that only the *first* set active bit of the frame must not occur in the last two cycles. Once a header arrives the value of the active bit does not matter for the remainder of the frame. We have not formally proven this however. Our more restrictive assumption was simpler to state and verify. The way the element is currently used ensures that it is not invalidated.

The theorem follows relatively easily from the correctness statement for `FAB4B4`. `FAB4B4` use slightly different notions of frames. This situation is similar to that encountered in the proof of `ARBITRATION`. The signals `ackOutEXT`, `dOutEXT` and `last` are considered separately. For each the proof is split into cases on the type of frame and the intervals within a frame.

```
⊢ FOR ackInEXT ackOutEXT ::(PSIGLEN 4).
    FOR dOutEXT dEXT ::(PSIG2LEN 4 8).
      ∼ (frameStartEXT 0) ∧
      ∼ (frameStartEXT 1) ∧
      (∀ t.
        ∼ (frameStartEXT (t + 2)) ∧ ∼ (frameStartEXT (t + 1)) ∨
        ∼ (EXISTSABIT I (Actives (dEXT t)))) ⊃
      FABRIC4B4 ((dEXT,frameStartEXT,ackInEXT),dOutEXT,ackOutEXT) ⊃
      (LOCAL last ::(PSIG2LEN 4 2).
        FABRIC4B4_SPEC DEFAULT_DATA last
          ((dEXT,frameStartEXT,ackInEXT),dOutEXT,ackOutEXT))
```


Acknowledgements

This report was generated from the original HOL source files using the mweb package of Wai Wong. The proofs relied heavily on his word library. This work was supported by SERC grant GR/J11133. I would like to thank Mike Gordon and Ian Leslie for their help and advice and for providing valuable comments on an earlier draft of this paper. I am also grateful to the members of the Automated Reasoning Group in Cambridge for proving a stimulating research environment. I am particularly indebted to Wai Wong and Brian Graham for their constant willingness to discuss problems that I encountered.

References

[1] Paul Curzon. The formal verification of the Fairisle ATM switching element: an overview. Technical Report 328, University of Cambridge Computer Laboratory, March 1994.

[2] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[3] Wai Wong. Modelling bit vectors in HOL: the word library. In *Proceedings of the 1993 International Workshop on Higher Order Logic Theorem Proving and its Applications*, 1993.