University of Southern California

CENG Technical Report: 01-03

Asynchronous 1 of n logic using single-track protocol

Student: Marcos Ferretti (ferretti@usc.edu)

Professor: Peter A. Beerel (pabeerel@usc.edu)

July 12th 2001

Asynchronous 1 of n logic using single-track protocol

A. Introduction

The proposed circuitry implements an asynchronous 1 of n logic using a single-track protocol. Single-track protocol was previously used to communicate control signals between logic blocks.

In our approach, two logic blocks communicate using the following procedure: the sender writes the data in the connecting wires, the receiver reads the data and reset the wires (restore them to a "blank" value), which informs the sender that the receiver got the data. While the wires are "blank", there is no data flowing and while the wires are holding data ("busy"), the sender will wait until the receiver set them "blank" before send a new data.

The main advantage of this idea is that there are no extra control wires and the data itself is performing the "hand-shaking" protocol between the logic blocks.

B. Description.

- Single track

The single-track protocol consists basically in specifying the "transmitter driver direction" and the "receiver restore direction". For example, for control signals, we can define that, if the communication wire was low ("ready"), the transmitter can drive the wire high and, when the receiver get the "request", it will "acknowledge" it by driving the wire low (restoring the wire level for the next communication).

The figure below shows an example of a block diagram for this procedure.

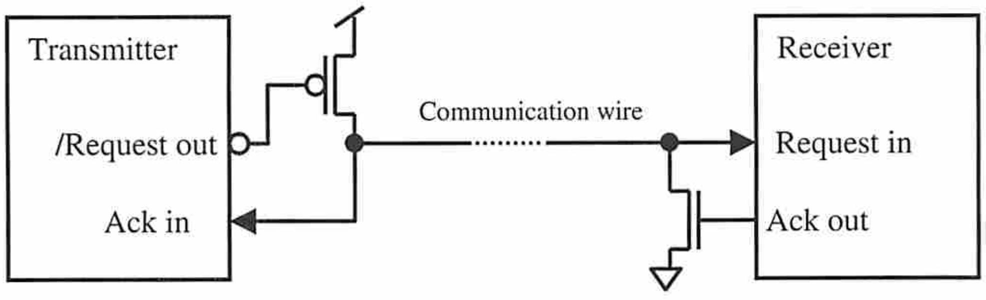


Figure 1 - Single-track protocol block diagram.

Notice that, it is also possible to implement "transceivers" using the communication wire to transport request and acknowledge in both directions if, for every communication event, it is well defined which block will send and which will receive. Also, with care, several transmitter and receivers may be connected to the same wire.

- **Wire voltage holders (staticizers)**

If the wire is subject to long periods of inactivity, a small memory unit (two inverters) may be used to "hold" the voltage level of the wire (see Figure 2-a). This solution implies that the transmitter and the receiver circuits fight against the state of the inverter, which may be energy inefficient.

We have a new approach to prevent the wire voltage drift from its intended value. The idea is that: the side that "drives high" will "hold low" and the side that "drives low" will "hold high". This "no-fight" solution offers the same drift protection of the two inverters approach and avoids short-circuit currents as shown in the diagram below.

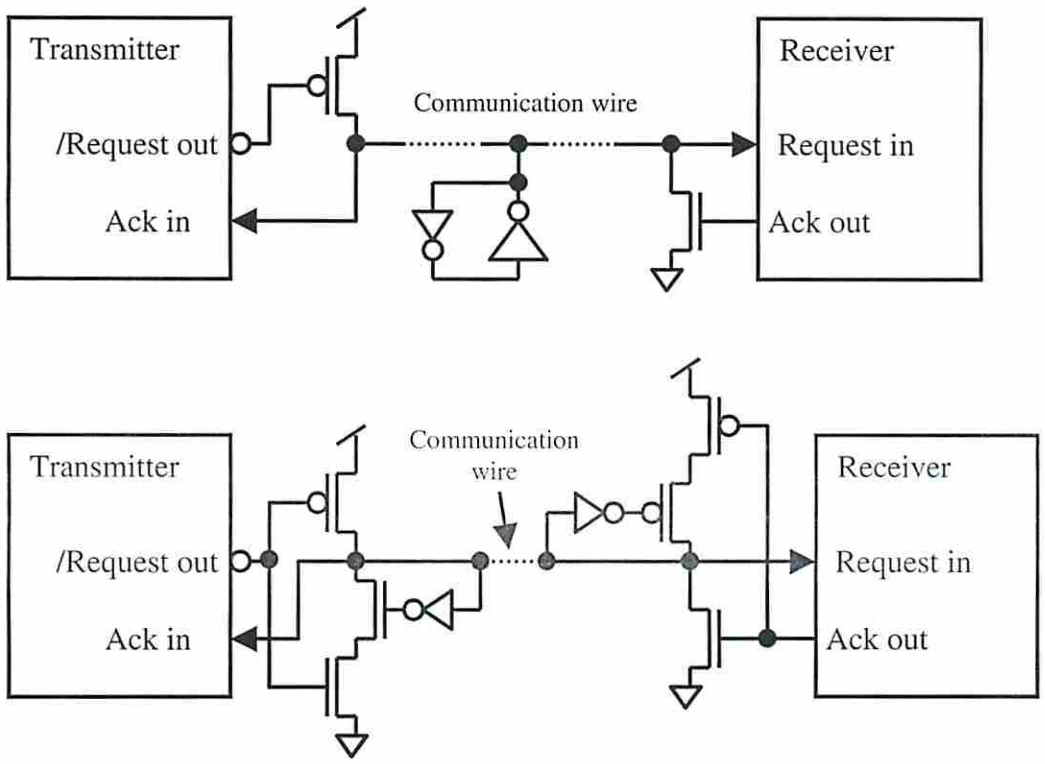


Figure 2 - Single-track protocol block diagrams with voltage holders (staticizers): (a) two inverters holder and (b) our "no-fight" holders.

For simplicity, the following diagrams don't show any line staticizer circuit, just the driver transistors.

- **The proposed 1 of n logic with single-track protocol**

In order to present the proposed asynchronous 1 of n logic using single-track protocol circuits, this documentation will focus on the 1 of 2 case, also known as "dual-rail" logic.

- **Dual-rail single-track buffer**

In asynchronous design, buffers are used to hold the information allowing "slack" matching or simply storing the data.

The figure below shows our dual-rail STFB (Single-Track Full Buffer) buffer implementation. L0 and L1 are the left-side dual-rail inputs and R0 and R1 are the right-side dual-rail output.

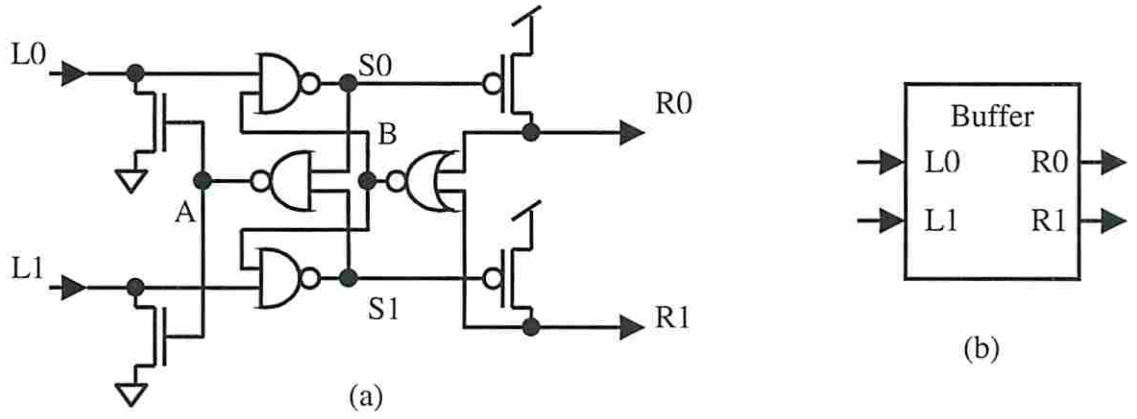


Figure 3 – Dual-rail (1of 2) STFB (Single-Track Full Buffer) Buffer: (a) logic diagram and (b) symbol.

A CMOS transistor-level implementation of Figure 3 is shown below.

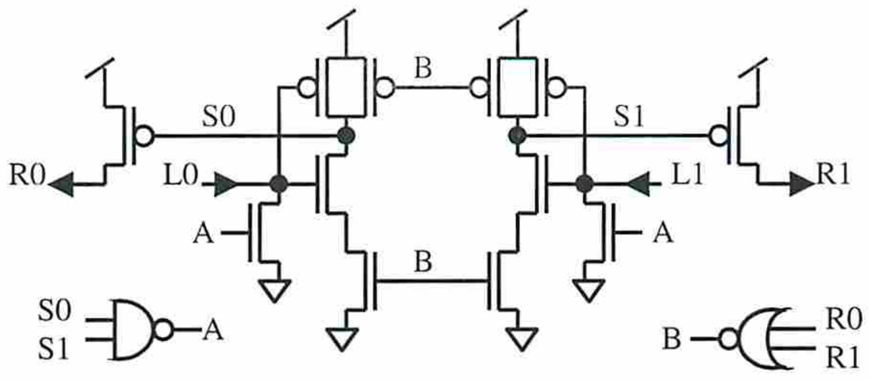


Figure 4 – Dual-rail (1of 2) STFB (Single-Track Full Buffer) Buffer transistor-level logic diagram.

Notice that, the internal signals A and B are generated based on the buffer internal state S0 and S1 and on the buffer outputs R0 and R1 respectively. When one of the internal state S0 or S1 goes low, the NAND generates A=1, which is used to reset the inputs "**A**cknowledging" the left environment. If R0 or R1 are high, the NOR generates B=0, which means that the output is "**B**usy" and any new incoming data must wait.

The figure below shows an optimization of the circuit shown in Figure 4. The NAND gates are merged and only the B signal restores S0 and S1.

A weak PMOS may be added to hold S0 and S1 high during long periods of inactivity. Also, reset signals (Reset and /Reset) may be added to initialize the input lines and the buffer state (S0 and S1).

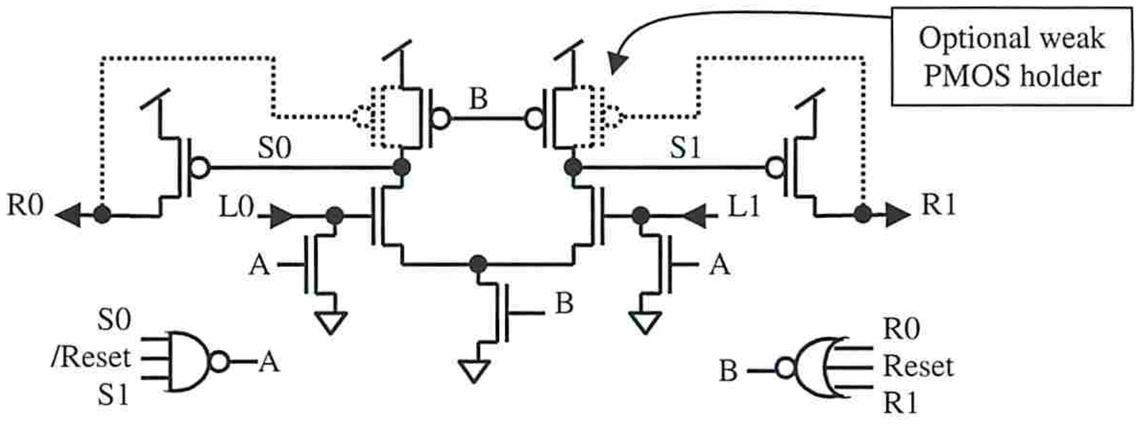


Figure 5 – Optimized Dual-rail (1of 2) STFB (Single-Track Full Buffer) Buffer transistor-level logic diagram with PMOS holder and reset inputs.

For simplicity, the optional reset inputs and the holder transistors will not be shown in most of the following circuits.

We can say that we have a SCD (State Completion Detector) to generate “A” (Acknowledge) when the data computation is “inside” the circuit and a RCD (Right environment Completion Detector) to generate “B” (Busy) when the output is still holding data (not ready). These two blocks are the essence of this new communication protocol. They supply locally the required signals for a proper communication avoiding external or explicit communication signals.

In the circuits of Figure 3, Figure 4 and Figure 5, we can see that the forward latency is 2 gate-delays, the backward latency is 4 gate-delays and the cycle-time is 6 gate-delays.

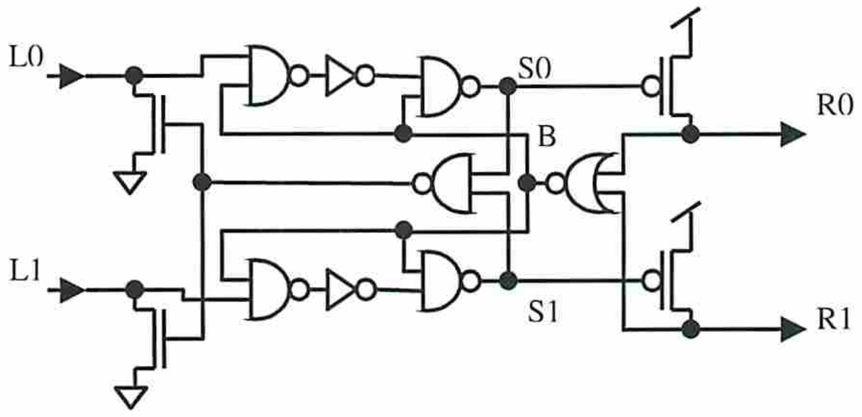
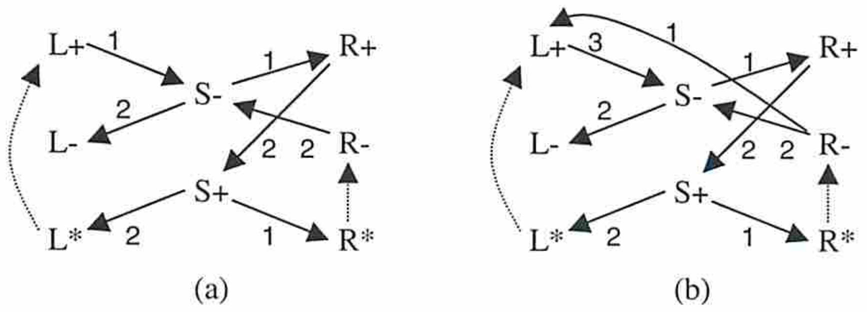


Figure 6 – Dual-rail (1of 2) STFB (Single-Track Full Buffer) Buffer logic diagram with timing margin of 2 gate-delays.

Also, the timing margin between the high driver “request” and the low driver “acknowledge” is zero. This may not be a problem after careful transistor sizing and timing analysis, but a circuit with explicit timing margin of 2 gate-delays is proposed below.

In Figure 6, we can see that the forward latency is 4 gate-delays, the backward latency is 4 gate-delays and the cycle-time is 8 gate-delays.

The timing margin between the high driver "request" and the low driver "acknowledge" is 2 because each line-driver transistor is still active for only 3 gate-delays.



(c) $STFB \equiv [[L+ \wedge R-]; R+, L-; R^*, L^*]$

Figure 7 – Signal Transition Graph (STG) of the buffer (a) from Figure 5 and (b) from Figure 6 and (c) STFB handshaking expansion (HSE).

Figure 7 shows two Signal Transition Graphs (STG) and the handshaking expansion (HSE) equation from the presented buffers. The signs "+", "-" and "*" represent the rising, falling and floating state of the signals respectively. The dashed arrows are driven by the left and right environments.

- **1 of n single-track buffer**

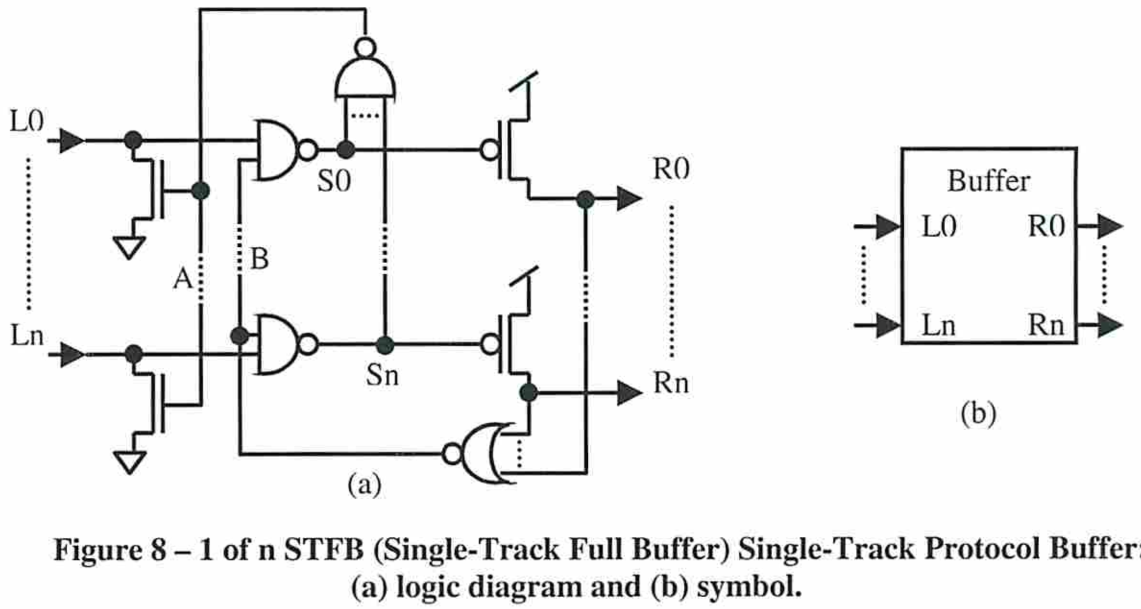


Figure 8 – 1 of n STFB (Single-Track Full Buffer) Single-Track Protocol Buffer: (a) logic diagram and (b) symbol.

The Figure 8 shows an example of our 1 of n STFB (Single-Track Full Buffer) buffer implementation. L0 to Ln are the left-side 1 of n inputs and R0 to Rn are the right-side 1 of n output.

- **Dual-rail single-track logic.**

Logic gates can be implemented to operate under this single-track protocol. For simplicity, the following examples are two inputs dual-rail (1 of 2) single-track logic gates. More inputs and/or 1 of n data can be naturally expanded.

- **Dual-rail single-track AND**

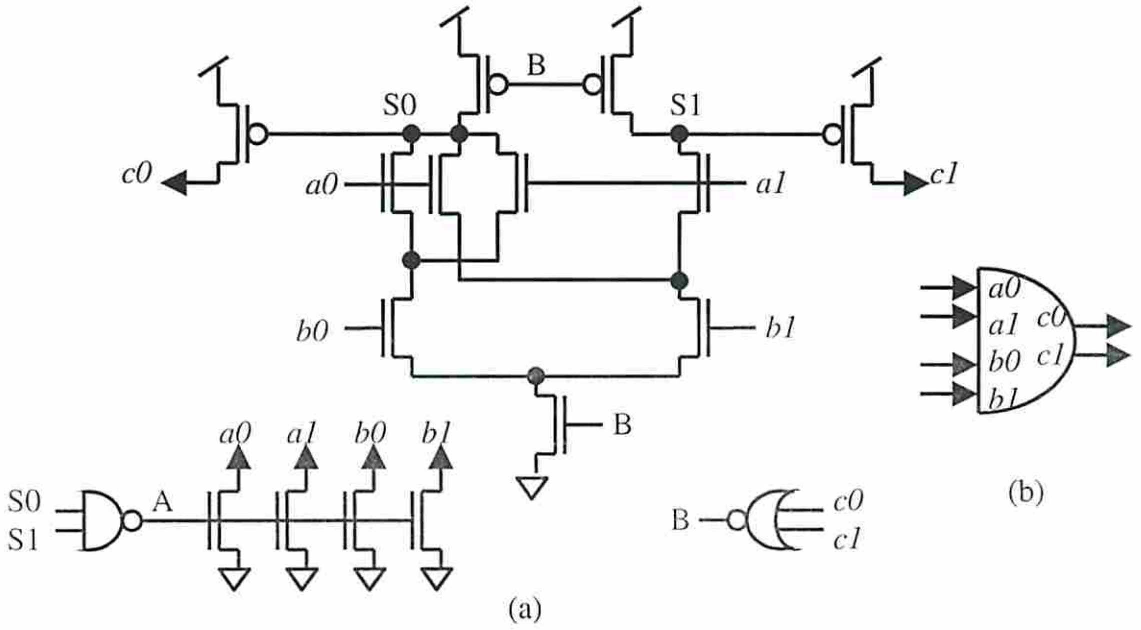


Figure 9 – Two inputs Dual-rail STFB_AND logic gate: (a) transistor-level logic diagram and (b) symbol.

In Figure 9, the circuit performs the AND logic operation: $a*b = c$, where a and b are dual-rail single-track inputs and c is the dual-rail single-track output.

All the inputs are "acknowledged" by the signal A when S0 or S1 goes low. For S1, this will happen when *a1* and *b1* are high. For S0, just *a0* or *b0* high should be enough to compute the logic result, but we need to be sure that both values (*a* and *b*) arrived at the logic gate before drive S0. That's why the input combinations: 00, 01 and 10 are covered to drive S0.

Notice that, since we are using dual-rail, to perform a NAND operation, we can just invert the output wires *c0* and *c1*.

The figure below shows a modified AND circuit. This circuit will forward a zero result if one of the inputs is zero even if the other input has not arrived yet. When all the inputs are finally present, they are acknowledged.

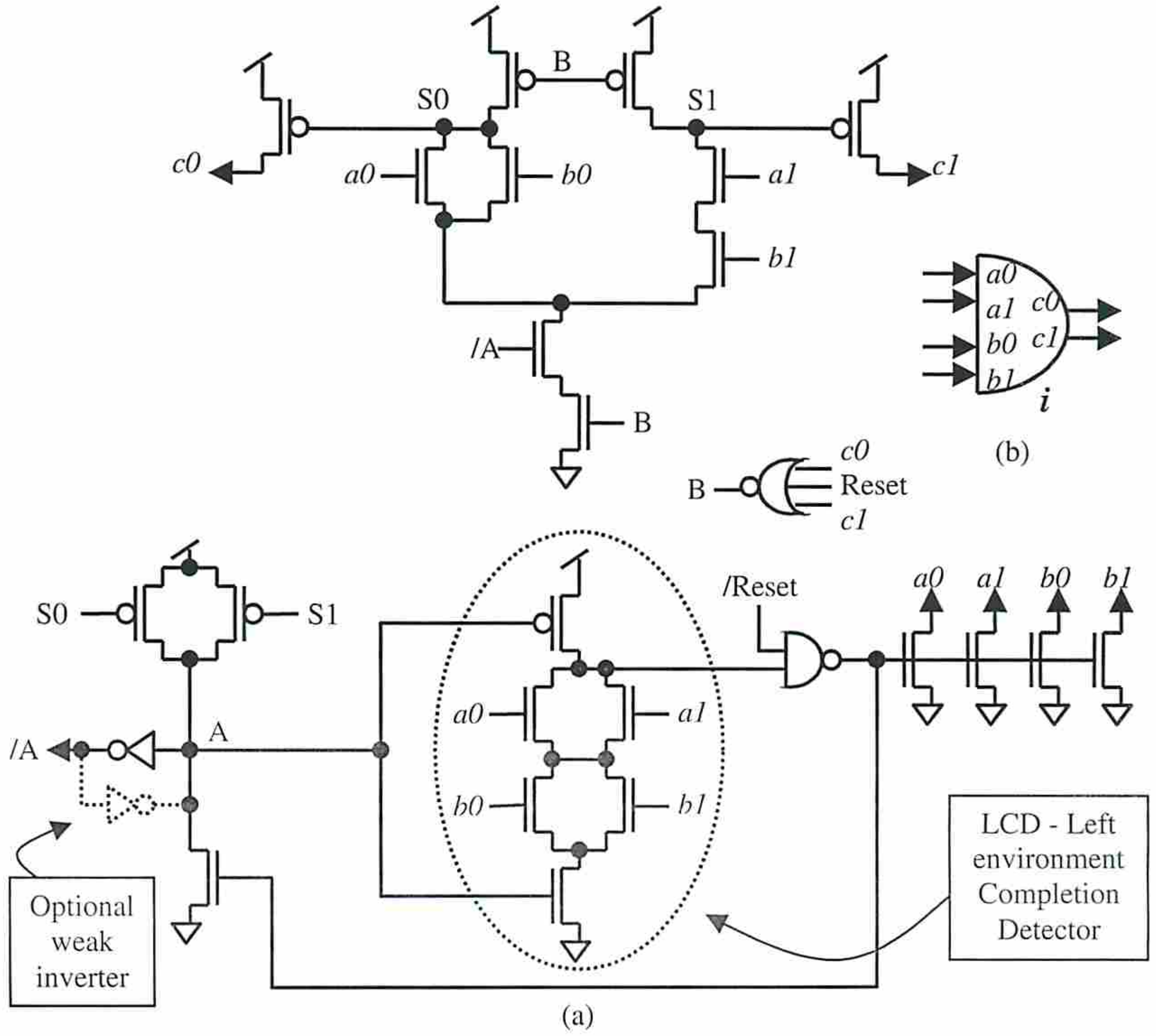


Figure 10 – Improved two inputs Dual-rail STFB_ANDi logic gate: (a) transistor-level logic diagram with reset signal and (b) symbol.

In Figure 10, while forwarding an "obvious result", the gate's SCD (State Completion Detector) sets A=1 which will disable the logic for future evaluations through /A=0 and will hold the information that an acknowledge is pending. When the LCD (Left environment Completion Detector) detects that all data are present, the acknowledge signal is passed to the transistors that will "consume" the data at the inputs and A is reset to zero. This will restore /A=1 and the gate will be ready to evaluate again.

The LCD may be slightly simplified since we know that the logic gate will only generate A=1 if certain inputs are already present. In this figure, for clarity, the LCD designed checks all inputs combinations.

Notice that, unlike other asynchronous designs, the LCD is used here to enable the acknowledge to the left environment not to enable the evaluation or pre-charge of the gate.

If the gate is subject to long periods of inactivity, a weak inverter may be added to hold the acknowledge state (A) as shown in the Figure 10.

- Dual-rail single-track OR

The following circuit perform the OR logic operation: $a+b = c$, where a and b are dual-rail single-track inputs and c is the dual-rail single-track output.

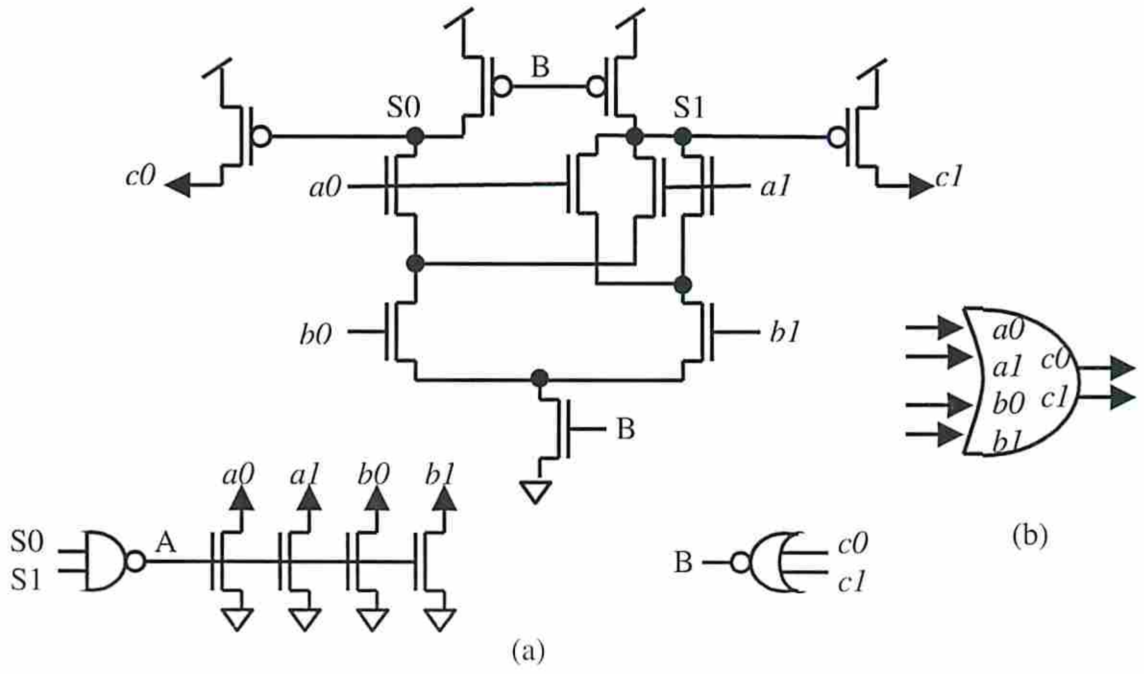


Figure 11 – Two inputs Dual-rail STFB_OR logic gate: (a) transistor-level logic diagram and (b) symbol.

In Figure 11, all the inputs are "acknowledged" by the signal A when S0 or S1 goes low. For S0, this will happen when *a0* and *b0* are high. For S1, just *a1* or *b1* high should be enough to compute the logic result, but we need to be sure that both values (*a* and *b*) arrived at the logic gate before drive S1. That's why the input combinations: 11, 01 and 10 are covered to drive S1.

Notice that, since we are using dual-rail, to perform a NOR operation, we can just invert the output wires *c0* and *c1*.

The figure below shows a modified OR circuit. This circuit will forward a one result if one of the inputs is one even if the other input has not arrived yet. When all the inputs are finally present, they are acknowledged.

In Figure 12, while forwarding an "obvious result", the gate's SCD (State Completion Detector) sets A=1 which will disable the logic for future evaluations through /A=0 and will hold the information that an acknowledge is pending. When the LCD (Left environment Completion Detector) detects that all data are present, the acknowledge signal is passed to the transistors that will "consume" the data at the inputs and A is reset to zero. This will restore /A=1 and the gate will be ready to evaluate again.

The LCD may be slightly simplified since we know that the logic gate will only generate A=0 if certain inputs are already present. In this figure, for clarity, the LCD designed checks all inputs combinations.

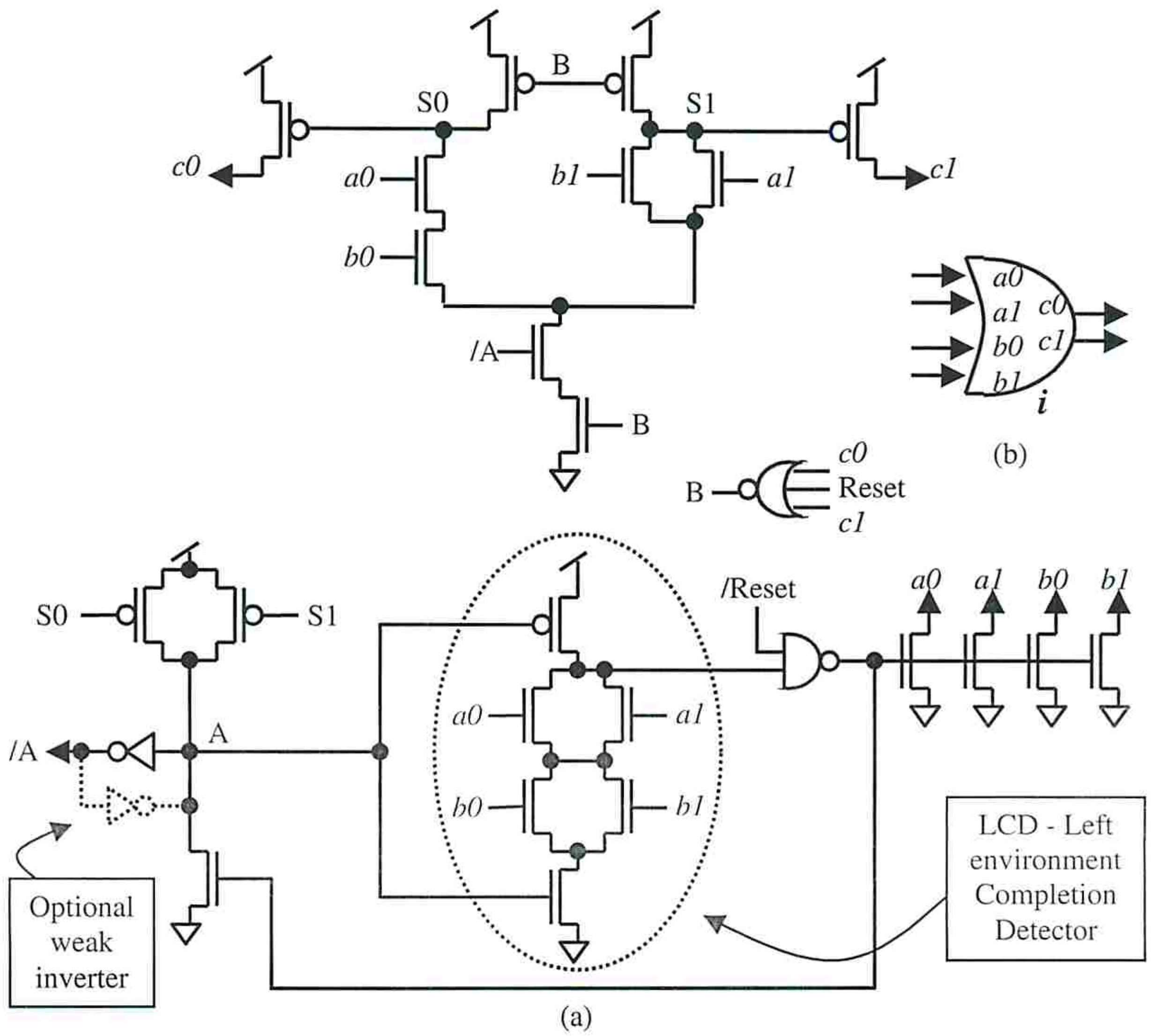


Figure 12 – Improved two inputs Dual-rail STFB_ORi logic gate: (a) transistor-level logic diagram with reset signal and (b) symbol.

Notice that, unlike other asynchronous designs, the LCD is used here to enable the acknowledge to the left environment not to enable the evaluation or pre-charge of the gate.

If the gate is subject to long periods of inactivity, a weak inverter may be added to hold the acknowledge state (A) as shown in the Figure 12.

- Dual-rail single-track XOR

The following circuit perform the XOR logic operation: $a \oplus b = c$, where a and b are dual-rail single-track inputs and c is the dual-rail single-track output.

In Figure 13, all the inputs are "acknowledged" by the signal A when S0 or S1 goes low. For S0, this will happen when a is equal to b, which means: *a0* and *b0* are high or *a1* and *b1* are high. For S1, *a* must be different than *b*, which means: *a1* and *b0* high or *a0* and *b1* high

Notice that, since we are using dual-rail, to perform a XNOR operation, we can just invert the output wires *c0* and *c1*.

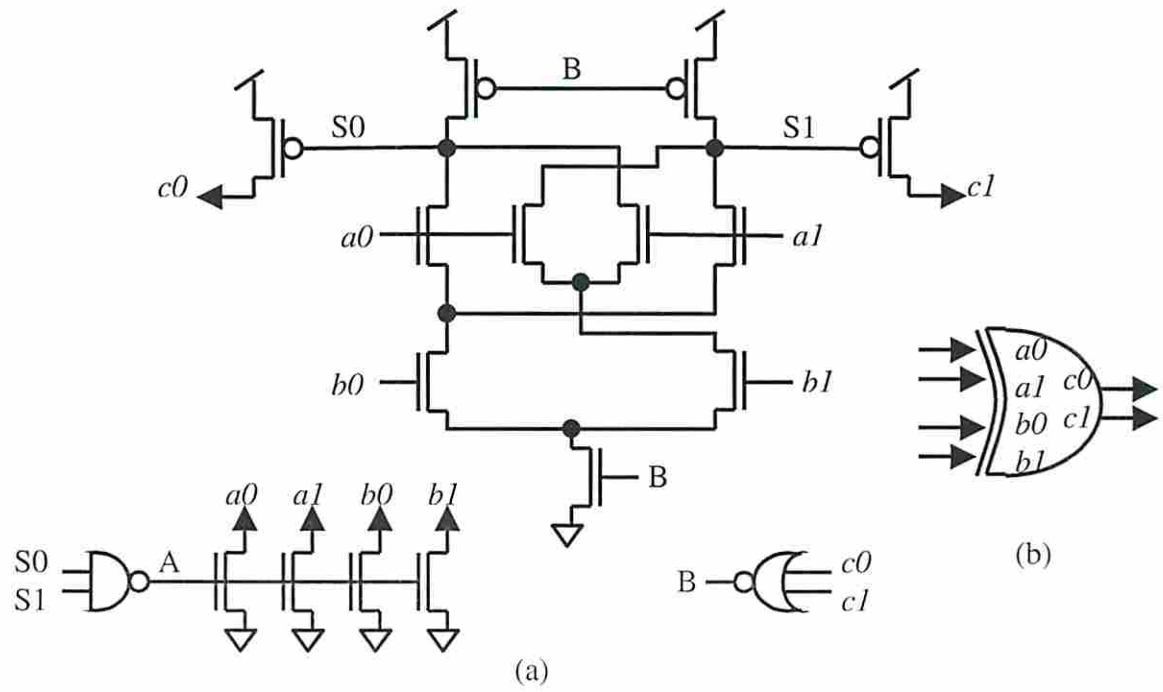


Figure 13 – Two inputs Dual-rail Single-Track XOR logic gate: (a) transistor-level logic diagram and (b) symbol.

Unlike the AND and the OR gates above, the XOR must wait until all its inputs are present to generate the proper result. Therefore, no improved XOR gate can be designed.

- **Dual-rail single-track Fork operation**

The Fork operation consists in copy the incoming data to several different paths if all output paths are ready. If one or more paths are busy, the data must wait.

The figure below shows the 1 to 2 fork circuit.

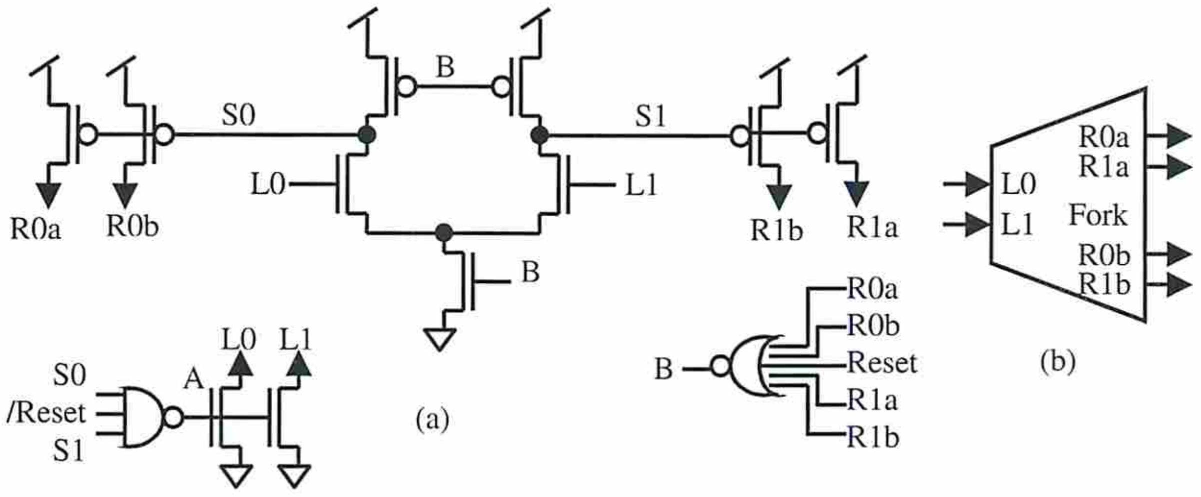


Figure 14 – Dual-rail STFB 1 to 2 FORK Buffer: (a) logic diagram and (b) symbol.

- **Dual-rail single-track Non-conditional Merge operation**

The Non-conditional Merge (also know as Join) operation consists in concatenate the incoming data from several different paths if the output path is ready. If the output path is busy, the data must wait. The inputs are mutually exclusive by the left environment.

The figure below shows the 2 to 1 non-conditional merge circuit and symbol.

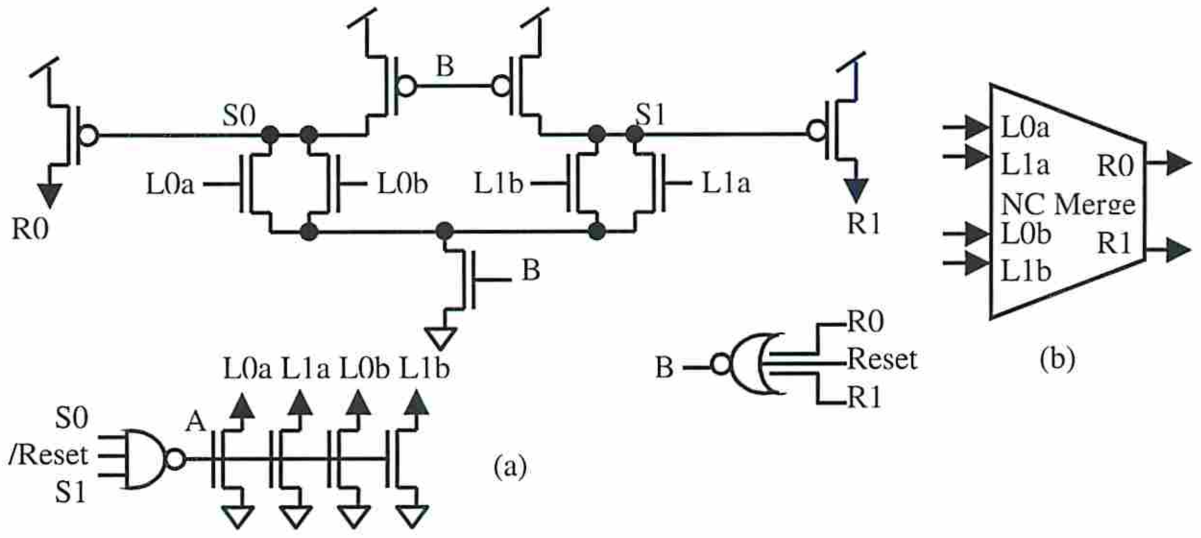


Figure 15 – Dual-rail STFB 2 to 1 NC MERGE Buffer: (a) logic diagram and (b) symbol.

- **Dual-rail single-track Split operation**

The Split operation consists in forward the incoming data based on a control (C) input. If the chosen output path is busy, the data must wait. After forwarding the data, the control signal is also consumed.

The figure below shows the 1 to 2 split circuit and symbol. In this example, when C=0, L is directed to Ra and, when C=1, to Rb.

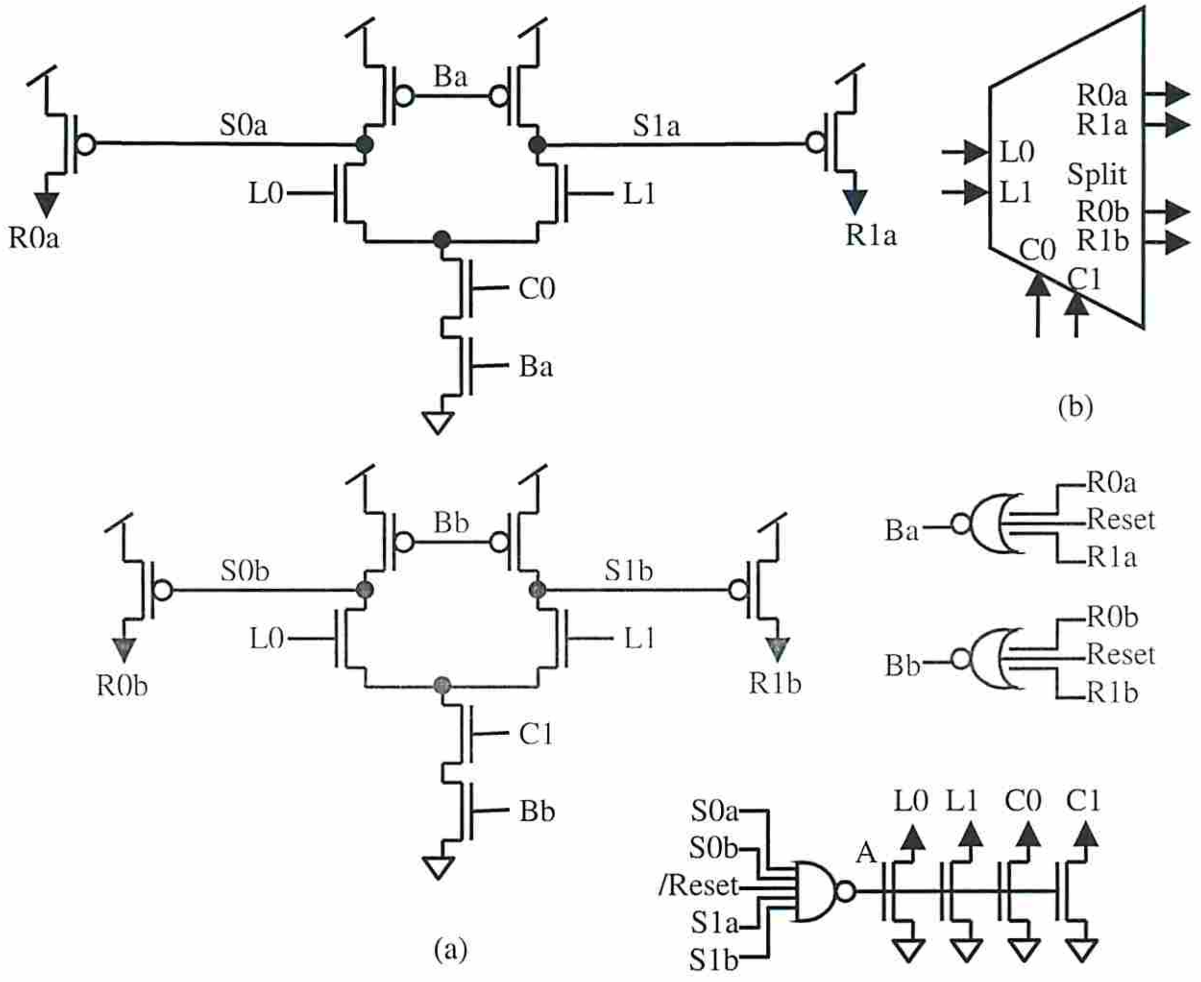


Figure 16 – Dual-rail STFB 1 to 2 SPLIT Buffer: (a) logic diagram and (b) symbol.

- **Dual-rail single-track Merge operation**

The Merge operation consists in choosing one of the incoming data based on a control (C) input. If the output path is busy, the data must wait. After forwarding the data, the control signal is also consumed.

The figure below shows the 2 to 1 merge circuit and symbol. In this example, when C=0, La is directed to R and, when C=1, Lb is directed to R.

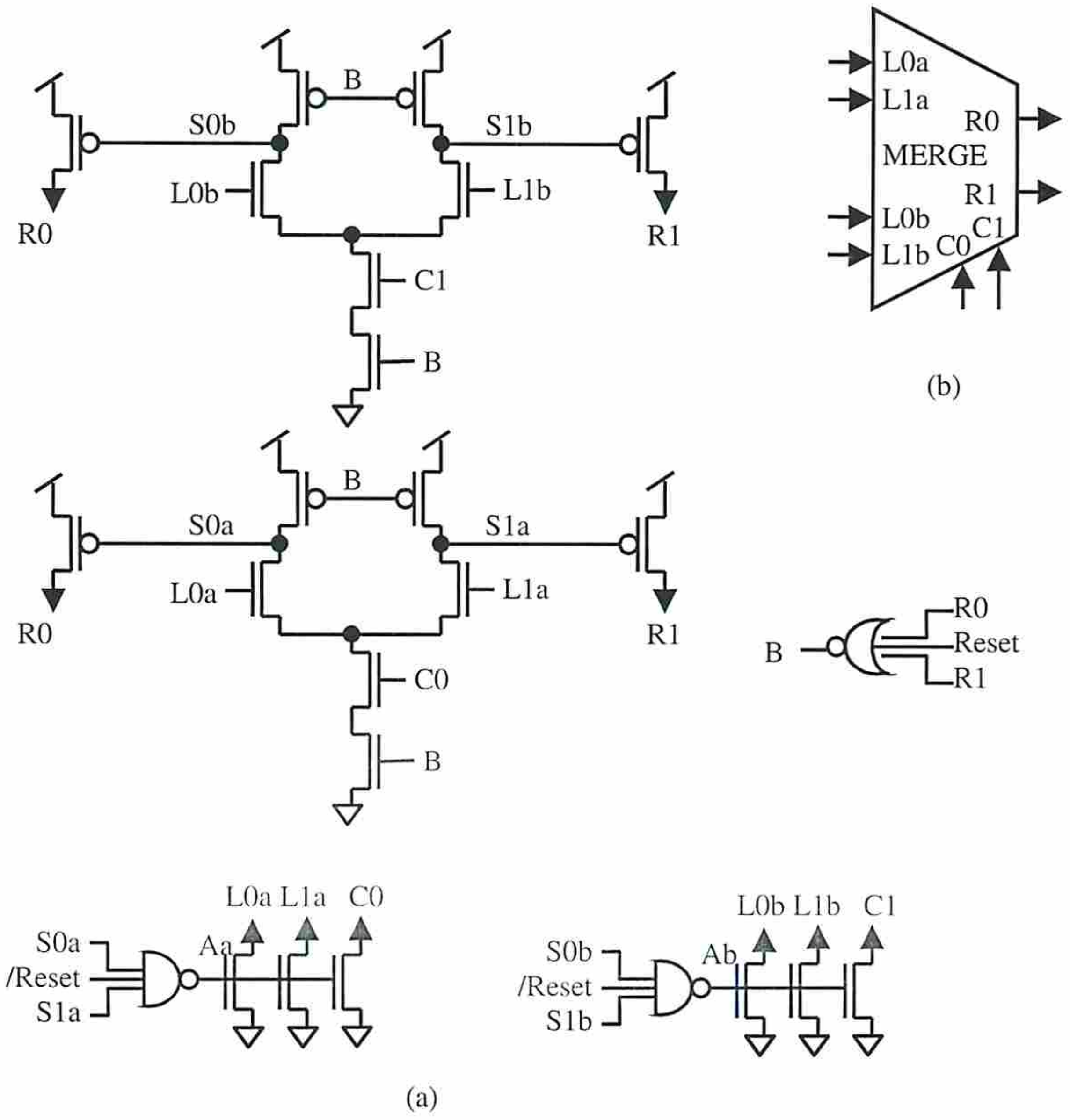
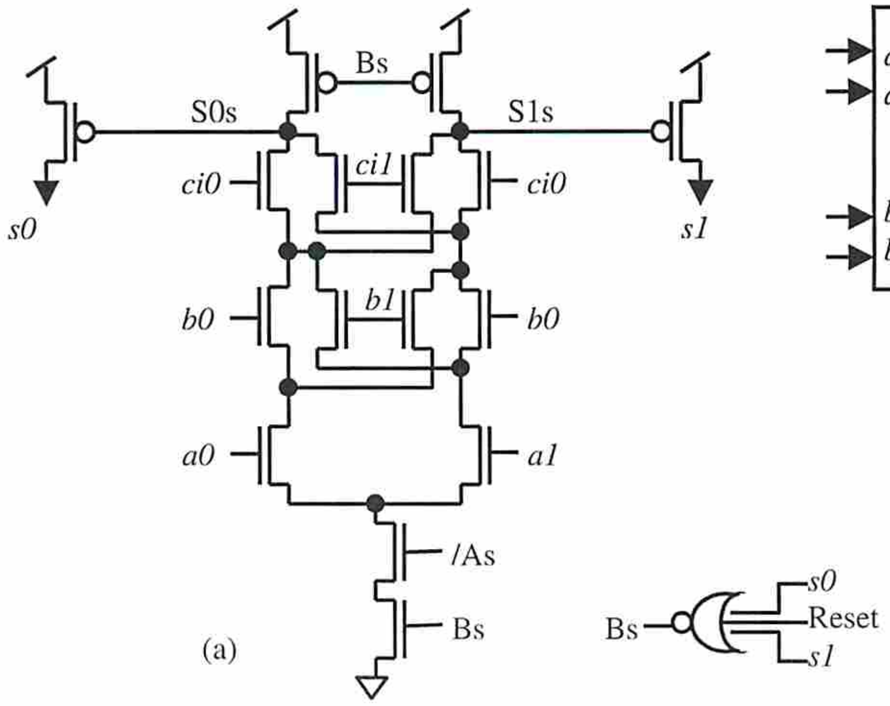


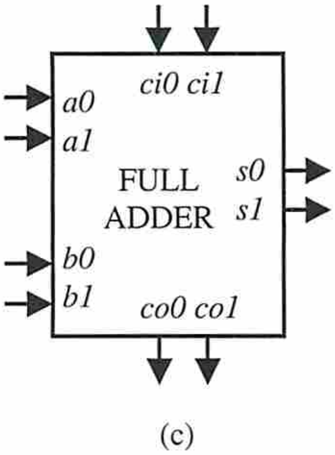
Figure 17 – Dual-rail STFB 2 to 1 MERGE Buffer: (a) logic diagram and (b) symbol.

- **Dual-rail single-track Full Adder (FA)**

To implement a full adder (STFB_FA) we need to compute the sum and the carry out before reset the inputs as shown below.

A three-input XOR and a three input majority (MAJ) gate basically compose the full adder. The XOR generates the sum ($s=a+b+ci$) and the MAJ generates the carry out (co=MAJ(a,b,ci)).





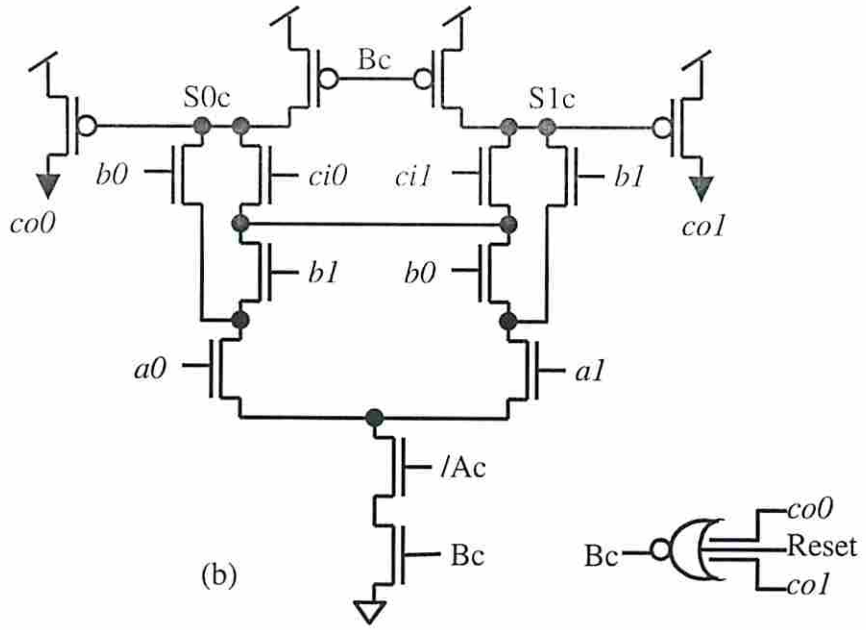


Figure 18 – Dual-rail STFB_FA main blocks: (a) three-input XOR gate to generate the sum, (b) three-input MAJ gate to generate the carry out and (c) symbol.

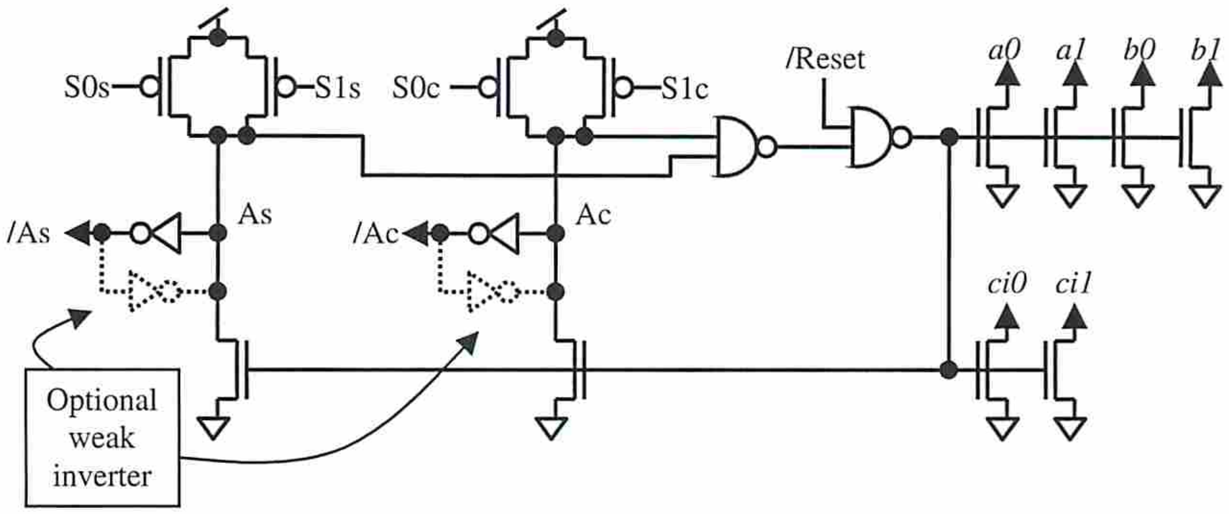


Figure 19 – Dual-rail STFB_FA acknowledge circuitry.

In Figure 18 and Figure 19, we have the proposed STFB_FA circuits. In this implementation, if *a*=1 and *b*=1, we have enough information to generate a carry out *co*=1 and, if *a*=0 and *b*=0, we have enough information to generate a carry out *co*=0. This will speed-up the result and the circuit of Figure 19 will acknowledge all the inputs when both results (*s* and *co*) have been transmitted.

Notice that, the same result would be possible if we duplicate the input signals (*a, b, ci*) using three STFB_FORK gates and, then, connect to one three-input XOR gate and one three-input MAJ gate. The main difference in this case is that we add the FORK gate in the critical path (the carry path), which may result in a lower performance when compared with the proposed circuit in Figure 18 and Figure 19.

- **Dual-rail four-phase to single-track transmitter (Tx)**

In order to interface with four-phase asynchronous logic, the circuit below is an example of a transmitter four-phase to single-track buffer.

In this circuit, if Le is high and the right environment is ready, a data from the left environment will be transmitted and the signal Le will be set low. This also disables the buffer, avoiding re-transmit the same data after the right environment consumes it.

Le will remain low until both inputs return to zero (four-phase protocol). When this happens, Le is set high and the transmitter is ready for the next data.

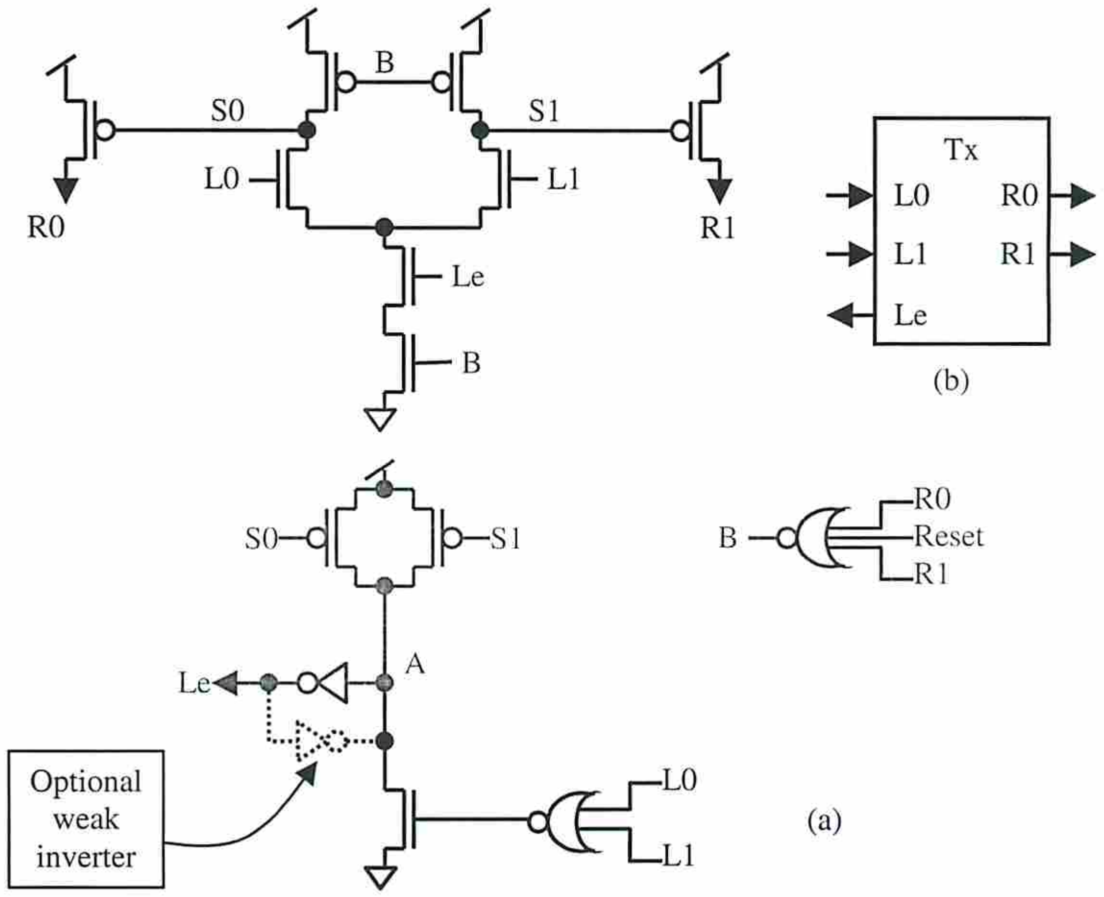


Figure 20 – Dual-rail (1of 2) STFB_Tx Buffer: (a) logic diagram and (b) symbol.

- **Dual-rail single-track to four-phase receiver (Rx)**

In order to interface with four-phase asynchronous logic, the circuit below is an example of a receiver single-track to four-phase buffer.

In this circuit, if Re is high (the right environment is ready), a data from the left environment will be received and the buffer will wait for the signal Re to be set low.

When Re goes low, a three gate-delay pulse is generated to consume the left environment data and the receiver is reset (R0 and R1 goes low).

While Re is low, R0 and R1 are reset and no new data is received (four-phase protocol). When Re returns to high, the receiver is ready for the next data.

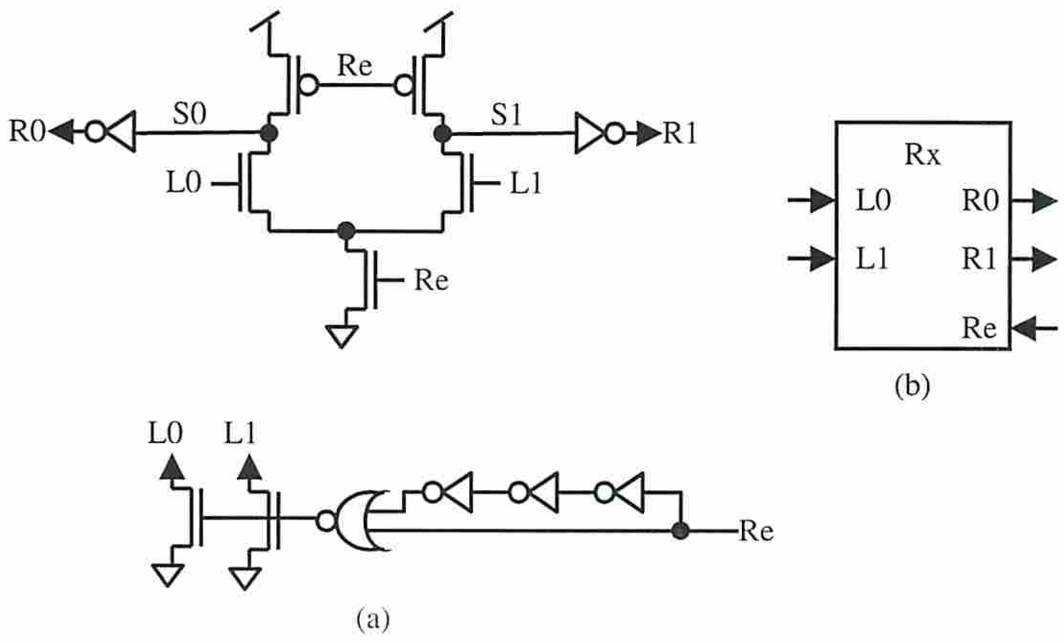


Figure 21 – Dual-rail (1of 2) STFB_Rx Buffer: (a) logic diagram and (b) symbol.

- **Dual-rail single-track Data Consumer (DC)**

In this single-track protocol, a data must be consumed or it will remain in the "wires" and will prevent the previous block to continue computing.

The circuit below simply destroys unwanted data by detecting it and resetting it.

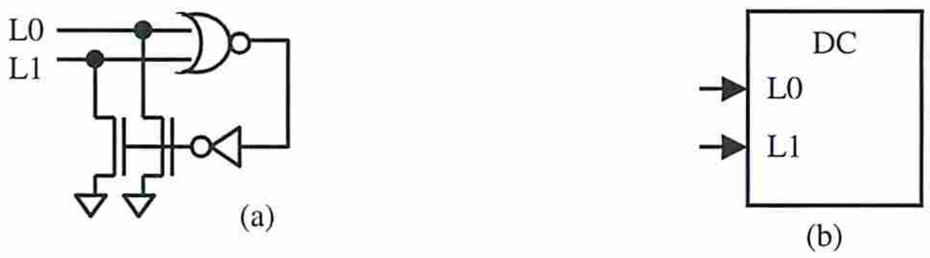


Figure 22 – Dual-rail STFB_DC: (a) circuitry and (b) symbol.