FORTH-ICS / TR-050 June 1992

Design of a High-Speed UART VLSI Library Cell

Panagiota Vatsolaki

Abstract

We present the sampling and decoding algorithm and the VLSI implementation of a high-speed UART (Universal Asynchronous Receiver-Transmitter) library cell to be used in custom or semi-custom VLSI chip designs. Our approach to data recovery, which is based on signal preprocessing and an innovative decoding algorithm, operates with as few as 2 samples per bit time, thus achieving a high communication rate. Using a clock of frequency f MHz, this UART can transmit and receive at a rate of up to f Mbits/s, without any internal multiplication of the clock frequency.

The current design, that was submitted for fabrication operates at data rates up to 25 Mbits/s (ES2 1.5 μm CMOS standard cell technology), while extensive simulations for a higher performance technology (1 μm gate-array) verify that our cell operates at data rates up to 60 Mbits/s. We are currently performing post-fabrication testing, and some preliminary results show that the prototypes operate succesfully at the data rate of 20 MHz. The resulting cell is small and flexible, making it suitable to be used as a building block in chip designs. It can serve as an interface between serial asynchronous communication links, or as a building block for fast and inexpensive networks.

† Institute of Computer Science, F.O.R.T.H., Greece
E-mail: vatsola@csi.forth.gr

Abstract

We present the sampling and decoding algorithm and the VLSI implementation of a high-speed UART (Universal Asynchronous Receiver- Transmitter) library cell to be used in custom or semi-custom VLSI chip designs. Our approach to data recovery, which is based on signal preprocessing and an innovative decoding algorithm, operates with as few as 2 samples per bit time, thus achieving a high communication rate. Using a clock of frequency f MHz, this UART can transmit and receive at a rate of up to f Mbits/s, without any internal multiplication of the clock frequency.

The current design, that was submitted for fabrication operates at data rates up to 25 Mbits/s (ES2 1.5 μm CMOS standard cell technology), while extensive simulations for a higher performance technology (1 μm gate-array) verify that our cell operates at data rates up to 60 Mbits/s. We are currently performing post-fabrication testing, and some preliminary results show that the prototypes operate succesfully at the data rate of 20 MHz. The resulting cell is small and flexible, making it suitable to be used as a building block in chip designs. It can serve as an interface between serial asynchronous communication links, or as a building block for fast and inexpensive networks.

Acknowledgements

I wish to thank everyone who helped me with this work, and to the completion of my studies at the University of Crete. First of all, I express my gratitude to my supervisor Manolis Katevenis for his guidance, initial idea, support and valuable advice throughout my graduate studies.

I would also like to thank Professor A. Traganitis who participated in the initial discussions about this work and provided ideas and improvements. I also thank him, together with Professor C. Courcoubetis, for participating in my thesis' committee.

George Kalokerinos spent many days helping me in the experiments for determining expected signal distortion, and provided corrections in the greek version of this text. Also, George Dimitriadis was very helpful at difficult moments with the VLSI design software. I thank them both.

I would like to thank the Institute of Computer Science of the Foundation for Research and Technology–Hellas (FORTH) for the financial support provided during my studies. This work was supported by the ESPRIT ''AMUS'' research project (contract 2716). The software used in my work was provided through the ESPRIT ''EUROCHIP'' educational program (contract 3603), and the chip was sent for fabrication through that same program.

Last, I send my deepest thanks to my parents and my family for their love and support.

Table of Contents

Chapter 1

Introduction and Overview

In the area of serial asynchronous communications, the RS-232 physical layer protocol is the most frequently used standard, as implemented by the UART chips. These chips recover the transmitted data by oversampling the incoming serial stream with a fast clock. Therefore, the need for such a fast clock is the reason that limits their highest achievable speed of operation. We have designed a UART cell, based on purely digital techniques, that can be used in custom or semi-custom VLSI chip designs. The approach adopted in the design of our UART cell, was to recover the transmitted data by preprocessing of the serial input and a ‘‘smart’’ decoding algorithm. This method overcomes the limitations of the standard UART cells and has the advantage of being capable to operate at high frequencies. In this introduction, section 1.1 will state the framework of serial communications, and briefly describe the RS-232 interface standard. Section 1.2 discusses the main problem of serial communications, the clock recovery, and the usual methods for achieving it. Finally, Section 1.3 presents an overview of the UART cell that was designed.

1.1 Serial Communication

In the area of digital communications, bits of binary data are commonly transferred by changes in current or voltage. The two primary distinctions of the type of transfer on the physical medium, is between *serial* and *parallel* transmission. In *serial* communication, the transmitted information is carried over a single line, while in *parallel*, bits of data are sent simultaneously either over separate lines or on different carrier frequencies on the same communication line. Parallel data transfers are motivated by high speed requirements in short distances. However, as the distances between interconnected devices increase, the timing skew between the multiple signals becomes the critical factor. Also, the cost of multiple cabling and equipment must be considered: serial transmission achieves to reduce the number and cost of equipment needed. Another reason for using serial communications is the interfacing with available data transmission media, e.g. telephone lines. Since there is only one line available, data of parallel form are sent using time multiplexing. The receiver and transmitter pair serialize the bits that represent the data, send them over a single line, and reassemble them in parallel at the other end of the cable.

Since in digital communications, there exists timing skew between the data and clock signal, the synchronization between receiver and transmitter can not be done assuming that the two nodes run under the same clock. The existing synchronization schemes distinguish the data links in *synchronous*, where the clock is sent together with the data, and in *asynchronous*, in which the data do not carry any clocking information. In *synchronous* interconnect, the exact departure or arrival time of each bit of information is predictable, but in *asynchronous*, the data arrive to the receiver asynchronously to its local clock, at non-uniform rates. Moreover, synchronous mode of transmission is used for time-constrained applications such as voice and real-time traffic, while traffic without specific constraints is transferred in asynchronous mode. The majority of serial communication links in use are *asynchronous*.

In *asynchronous* transmission, no clock signal is sent with the data. For this reason, the transmitter and receiver must agree upon all the parameters of the bit format, including the nominal bit time, parity, number of data bits for each frame, and number of stop bits. The transmission is controlled by start and stop patterns at the beginning and the end of each data character. In short, protocols of serial asynchronous links operate as follows: When the transmitter is idle, the line is maintained in a continuous ‘‘mark’’ (idle) state. A data transmission may be initiated at any time by sending a *start* bit, followed by the bits of data, and finally the stop bit, as shown in figure 1.1. After that, the transmitter may immediately send a new start bit, if another character is available, or maintain the mark (idle) state as long as it is idle.

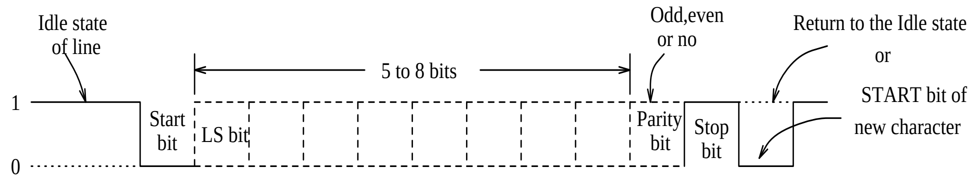


Figure 1.1: Asynchronous Data Character format.

The most common physical layer protocol for serial asynchronous communication is the RS-232 protocol. It was originally developed as a specification for connecting devices using the telephone network as an intermediate medium, through modems. Later, it became widely adopted to connect computers with ASCII terminals or computers with other computers through serial RS-232 ports. Its specification includes the electrical signal characteristics, the interface mechanical characteristics (connectors and interfacing cables), a functional description of the interchange circuits, and recommendations and explanatory notes. The RS-232 standard uses *bipolar modulation*, that is, it uses two voltage signs (polarities), in contrast to *unipolar modulation* which uses only one voltage sign. The bit 0 is represented by a voltage value *V* in the range [+3V, 25V], and bit 0 is represented by a voltage -*V*. The other main specifications of the protocol are:†

(1) Driver output impedance with power off should be less than 300 ohms.
(2) Driver output voltage with open drain less than 25V.
(3) Driver slew rate less than 30V/µs.
(4) Receiver output with +3V input should be space (logic 0)
(5) Receiver output with -3V input should be mark (logic 1).
(6) The capacitance of the driven circuit should not exceed 2500 pF, including the cable capacitance.

However, the above specifications are not strictly followed in the various implementations of the protocol. The reasons for the relaxed adherence to the rules are mainly the problems resulting from the electrical specifications. For example, the limit of the capacitance of the driven circuit can be met only for cable lengths up to 15 meters since common cables have capacitances of 50-100 pF/m.

The most popular and widely used implementations of the RS232 protocol is provided by the UART (Universal Asynchronous Receiver Transmitter) chips. Commercially available UART chips, offer bit rates from 50 b/s up to 153 Kb/s.

1.2 Existing Methods for Clock Recovery

In serial asynchronous communications, no clocking information is transmitted with the data, and no common clock is distributed to the receiver and transmitter. The clock recovery problem exists here in the sense of generating a clock with which the input serial stream should be sampled. The main approaches to the clock recovery are the use of Phase Locked Loops (PLL), and oversampling.

A Phase Locked Loop (PLL) is a circuit that synchronizes a periodic output signal (generated by an oscillator) with a reference input signal in frequency as well as in phase [Best85]. In the synchronized ‘‘locked’’ state, the phase error between the oscillator’s output signal and the reference signal is zero, or very small. If a phase error builds up, a control mechanism acts

Selected from the detailed protocol specification as it appears in [McNam88].

on the oscillator in such a way that the phase error is again reduced to a minimum. In such a control system, the phase of the output signal is locked to the phase of the reference signal. The functional blocks of a typical PLL are: a voltage-controlled or current-controlled oscillator, a phase detector, and a loop filter. PLLs are classified depending on the type of phase detector (PD) used. The most frequently used PDs are of *linear* type which are built from analog components, and *digital* type, which are built from digital components and operate only on binary signals. However, in a DPLL system that uses a digital PD, the rest of the components (filters, VCOs, CCOs) may be analog, and thus generate intermediate analog voltages. There exist also all-digital PLLs, which are built exclusively from digital components. These are slower than analog PLLs, and are used for low frequencies, in the range of KHz. The performance of a PLL is characterized by its lock range, that is the range of frequencies over which phase lock is achieved. Normally, the lock range is the operational range. The majority of the commercial PLLs are built in NMOS or CMOS technologies, and can be used at low frequencies, up to several MHz; PLLs built in TTL technologies operate up to about 25 MHz, while ECL circuits cover the range up to several GHz. In the recent literature, implementations of all-digital PLLs appear to require complicated hardware, and their operation range is limited up to several KHz, as in [HaPu91]. On the other hand, implementations of linear PLLs built in advanced CMOS processes (˜0.8 μm channel length), like the one appearing in [JoHu88], operate at higher frequencies, in the range of MHz's. Apart from requiring high design expertise and fine tuning, these designs occupy considerable silicon area.

Available implementations of the asynchronous serial communication protocol by UART chips, determine the value of incoming bits by sampling the serial bit stream with a sample clock; the bit value is determined by a unique sample taken with this sample clock. In order to get a ''safe'' sample, the sampling must occur at the middle of the bit, as far as possible from transitions near the bit boundaries. Thus, the problem of determining the bit values is equivalent to generating a sample clock with edges at the center of bits. The receiver generates this sample clock by using another, fast clock, of frequency K x data-rate; usually K=16. Figure 1.2 illustrates the timing of the sample clock generation for a primary clock of 4x data-rate (b), and 16x data-rate (d). As soon as a start bit is detected as a change of the serial input from 1 to 0 (in the example figure at t=t0), a down-counter is loaded asynchronously with $K/2$. When the counter reaches 0, a sample clock pulse is produced.

In the ideal case, the timing of the sample pulse should be at the center of the bit, but since the receiver clock runs asynchronously to the Serial Input, this does not usually happen. The distance between the sampling point and the center of the bit is the sampling error, and in the worst case, the maximum sampling error is equal to the receiver (fast) clock period. To limit this error, a high resolution of the receiver's clock is required, since the higher the clock frequency is relative to the baud-rate, the greater resolution of the Serial input is.

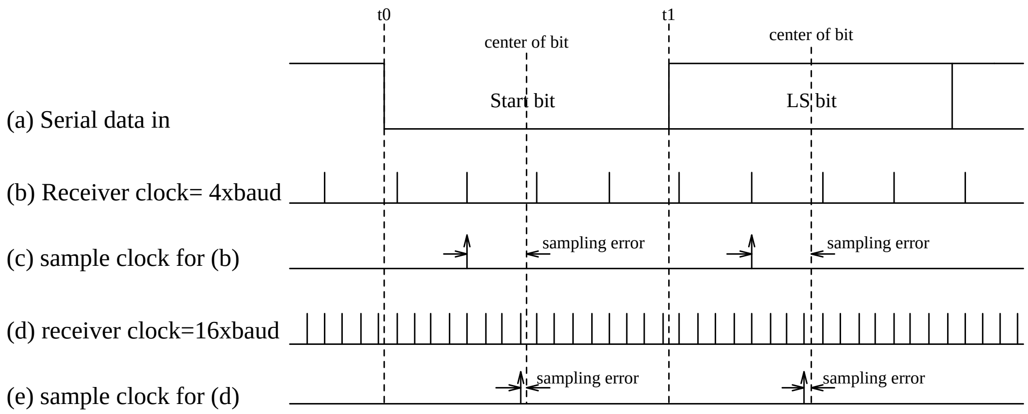


Figure 1.2: The effect of clock frequency on Sample clock generation in conventional UARTs.

Thus, conventional UARTs need fast clocks in order to determine with accuracy the center of the input bits, and their values. This fact ends up being limiting factor for their highest achievable speed of operation. As we move to higher speeds, the distortion of the input pulses gets worse, and the sampling error has to be very small in order to achieve correct results. So, the receiver clock should be many times higher than the data-rate. However, with the current technologies, realistic clock rates are up to 100 MHz; above this rate, many problems arise from the large interconnect delays and skews. Using this value, we can estimate the maximum communication rate to be 6.2 MHz or 3.2 MHz, for K=16 or 32, respectively.

1.3 Cell Overview

In the previous sections, we have seen the serial communications standards, and discussed about methods of clock recovery from a serial input stream. We have presented the commonly available implementations of asynchronous serial protocols, and their inefficiency for high speed operation. In order to overcome these limitations, we have designed a UART cell that employs purely digital techniques, and can operate at high rates based on a ‘‘smart’’ decoding algorithm. By preprocessing the serial input, and owing to its decoding algorithm, our cell eliminates the need for very fast sampling clock or complex analog components. Furthermore, the decoding algorithm copes with the signal distortion inherent at high frequencies, and enables the operation at high data rates. The designed cell is small and flexible, and can be easily used as a building block in custom or semi-custom chip designs.

Our UART cell can operate at any of the fixed baud rates of conventional UARTs, but can also send and receive data at much higher rates. Using an external reference clock of frequency

f_{clk}, it derives its internal transmit and receive clocks, setting the communication rate to any integer submultiple of twice that clock's frequency f_{clk}. More precisely, the communication rate can be set to $2f_{clk}/N$ bits/s, where N can take any integer value between 2 and 1,048,576 ($2^{20}-1$), resulting to a very wide communication range. For example, assuming an external clock of 25 MHz, the data rate can be set between 47 bits/s and 25 Mbits/s.

In order for the Receiver to run at high frequencies, we followed an approach different from that of traditional UARTs. As we have seen, conventional UARTs oversample the incoming data with a fast clock (usually 16x or 32x data-rate), and use the middle sample as the value of the bit, i.e. they neglect the samples near the bit boundaries, where transitions happen, and use only the ''safe'' sample from the middle of the bit. Our approach is to first shape the incoming signal through preprocessing filters, and then feed *all* acquired samples to an FSM that decides the bit value. The receiving algorithm implemented by the FSM can operate with very few samples per bit time, and thus avoids the need for a very fast sampling clock, and enables operation at high data rates. In the extreme case, we need only 2 samples per bit, which is the theoretical minimum to reconstruct a signal from its samples, as stated by the Sampling Theorem, [Op83].

Our cell was designed using primitive cells from the ES2 Standard Cell Library. The UART consists of two independent parts, the Transmitter and the Receiver. Each part of the UART was designed as a macrocell, in order to be used as a building block in custom or semi-custom VLSI chip designs. The transmitter constists of 650 gates, and occupies 1.8 mm^2, while the receiver consists of 850 gates, occupying 2.7 mm^2. We have included both the Transmitter and the Receiver into a chip, together with all necessary logic, to form a fully operational UART, that has been submitted for fabrication through EUROCHIP. This chip consists of 1.5 Kgates, occupying an area of 16 μm^2, including the I/O pad ring. It is packaged in a 28-pin DIL-type DIP, and powered with 0V and 5V supply voltages.

The current design, for the rather conservative 1.5 μm CMOS standard-cell process of ES2, was simulated to operate at 25 MHz, giving a data rate up to 25 Mbits/s. We have also tested the speed-up of our design, when implemented with a higher performance technology: using a differently optimized CMOS gate-array technology of 1.0 μm, we verified through extensive simulations that the UART operates at data rates up to 60 Mbits/s, using an external clock source of 60 MHz.

Preliminary tests of the fabricated devices verify that the chip operates at data rates up to 20 Mbits/s both in loopback mode and when two UART devices are interconnected through a 100 meter coaxial cable (see also see § 4.4 on testing) Tests at frequencies between 20 and 25 Mbits have not yet been performed.

Thus, instead of using oversampling that limits the operation range, or complex and area consuming analog techniques, we recover the clock using a decoding algorithm that tolerates distortion. In the next chapters, we will present the sampling and decoding algorithm, the VLSI implementation of the UART cell, the design methodology followed in the design and the testing of the fabricated devices, and conclusions and possible extensions of this work.

Chapter 2

The Sampling and Decoding Algorithm

In this chapter we present the sampling and decoding algorithm of the asynchronous receiver. The first section focuses on the pulse distortion patterns observed when transmitting at high frequencies, while the second section presents the preprocessing circuits and discusses how they modify the expected pulse distortion. In the next section after that, we present the decoding algorithm, and in the last section we derive its distortion tolerance.

2.1 Pulse Distortion Patterns

The method that conventional UARTs use for data recovery is oversampling the input serial signal with a fast clock, trying to accurately locate the middle of a bit pulse. At that point, they take a unique sample that determines the bit value. As we have seen in section 1.2, the key point for the operation of this method is the existence of an external clock which is much faster than the data rate. Hoewver, the need for such a fast clock proves to be the major drawback of this method, since it restricts the highest achievable operation speed, and practically sets its upper limit to the range of a few MHz.

In our work, we tried to overcome these limitations, and build a cell that provides the functional requirements of a standard UART, and furthermore, can communicate at much higher speeds. To accomplish our intentions, a completely different method than that of standard UARTs was adopted. First, the serial input signal is preprocessed through filters in order to counteract the distortion that the signal bears, due to the propagation through the physical medium. Then, the transmitted data are recovered in an algorithmic way: a finite state machine that implements the decoding algorithm decides on the value of the bits seen, taking into account all acquired samples. The receiving algorithm implemented by the FSM achieves to operate with very few samples, and thus enables the operation at high data rates. More specificly, the communication rate is determined through setting of the bit length parameter, N, expressed as an integer multiple of the half clock frequency. The communication rate is defined as $2f_{clk}/N$, where f_{clk} is the external clock frequency. The minimum value of N is 2, giving a data rate equal to the external clock frequency.

Given the main capabilities and overview of our cell, and before examining in detail its parts and operation, let us focus on the background of this work. A useful abstraction in the area of digital communications, is the notion of the *edge* or *transition* of a signal. This abstraction, that ignores the rise-time effects, holds in the cases where the rise times are very short in relation to the interval between edges. However, as clock rates increase, the signal behavior due to underlying physical phenomena, and ignored by this abstraction, becomes very important. A problem that is accentuated when we transmit at high frequencies is the distortion of the signal. This distortion is due to the non-linearities of the transmitter driver, to the transfer characteristics of the physical medium (different attenuation at different frequencies), and to the threshold point of the receiver sampler. The result is that the width of a received pulse is different from that of the transmitted one.

Initially, in order to study the frequency response of the RG58/U type coaxial cable, we ran spice simulations, where we transmitted digital square pulses at 50 MHz through a 100m cable. The coaxial cable was cut into 1m pieces, each of them modeled as a transmission line by a capacitor and an inductor. The values of these elements were the RG58/U coax cable parameters, that is, nominal impedance $z = 53.5 Ohms$, capacitance $C = 72.29 pf/m$, inductance $L = 0.35 \mu h/m$, and attenuation $4.65 db/100 ft at 100 MHz$. The transmitted pulse had a 5 ns rise time and a 5 ns fall time, and resulted to a received sinusoidal. Since the additive property holds for the signal amplitude, the compound effect of consecutive transmitted pulses is the sum of amplitudes of the partial signals. The ‘‘tails’’ of previous pulses add up to the current pulse, and depending on their polarity either amplify or lower the curve peaks. As a result, the pulses are ‘‘spreading’’ over the time axis.

As a second step we ran experiments where we found that pulses of one of the two binary values are favored by the transmission medium, the driver and the sampler, i.e. they expand, while pulses of the other binary value correspondingly shrink.

Figure 2.1 shows our experimental set-up. In this experiment, we transmitted digital, unmodulated, TTL-level pulses, at frequencies between 25 and 50 MHz, through 85-meter long coaxial cable of type RG58. With the use of an oscilloscope, we observed that the *strong*, high voltage pulses expanded by about 2 to 5 ns, at the expense of the low pulses. We measured the maximum change of a (single) pulse width to be 19% for the high pulses, and 27% for the low pulses. Since the total width of a long group of pulses must be the same before and after transmission, the expansion of one type of binary pulse should be at the expense of the other

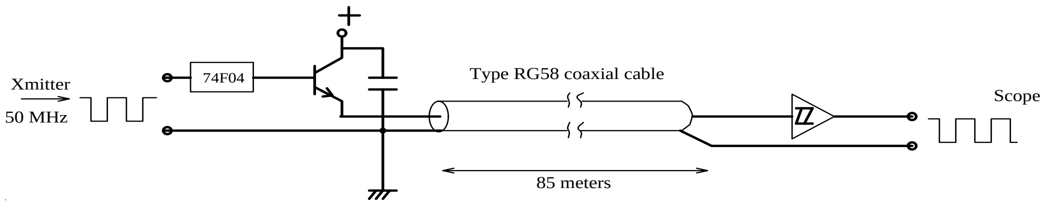


Figure 2.1: Experimental set-up for measuring signal distortion.

Digital pulses generated at frequencies between 25 and 50 MHz, go through FAST inverters in series (to enhance signal transitions), and are transmitted in a 50 Ohm coaxial cable (RG58) 85 meters long by a LH63 amplifier (for TTL to 50 Ohm conversion). The receiving amplifier was a FAST 74LS04 Schmitt trigger.

type, and the altering should be the same for the two types. The difference between the above two is due to the transmitted pulses being unbalanced. That is, the width of the high and the low pulses were not the same. This was due primarily to the tester that generated the patterns, and secondarily to the amplifier. If we relate the distortion to the effective frequency of the transmitted pulses, the the maximum distortion of the low pulses (27%) actually corresponds to a 60 MHz signal.

Table 2.1: Pulse width change for high and low pulses at frequencies 25-50 MHz

f MHz	T ns	transmitted 1-pulse width (ns)	transmitted 0-pulse width (ns)	received 1-pulse width (ns)	received 0-pulse width (ns)	1-pulse width distortion ns	1-pulse width distortion %
50	20	23	17	27.2	12.8	+4.2	18%
40	25	23.2	16.8	27.7	12.3	+4.5	19%
35.7	28	34.6	21.24	38.16	17.6	+3.56	10%
31.2	32	39.4	24.6	42	22	+2.6	6.5%
27.7	36	41.7	30.3	44.8	27.2	+3.1	7.4%
25	40	44.5	35.5	46.6	33.4	+2.1	4.7%

Table 2.1 summarizes our results. One can observe that the pulse distortion is worse for high frequencies. As we move to lower frequencies, the change of the pulse width gets smaller in absolute numbers, and so does the percentage of the width change. However, there are some irregularities in this transition: for the transmission frequency of 50 MHz we get better results than for the frequency of 41.6 MHz. This may be due to the correlation of some signal frequencies with the cable length, resulting from reflections.

When many pulses of the same value are transmitted next to each other, a single wider pulse results. Then, it becomes important to know whether the aggregate-pulse distortion is a function of the original clock frequency or of the total width of the ‘‘concatenated’’ pulse. We found that no clear and observable changes in the distortion of the short-pulses due to long-pulses: If we transmit the pattern "0111111" repeatedly, the 0-pulse distortion is the same as for the pattern "011" at the same base frequency. In summary, we found that high pulses at frequencies between 20 and 50 MHz expand by a rather constant value of 2 to 5 ns, and the low pulses shrink by the same amount. In general, we can say that digital pulses of one of the binary values expand while those of the other binary value correspondingly shrink. We will call the former *strong*, and the latter *weak* pulses.

As an example of signal distortion, let us assume that the bit length N =3, and examine how the bit sequence "10110" is transmitted, applying the above experimental results. The duration of a single pulse is defined as NT, where T is the half-clock period; thus, in our case, a bit is transmitted as a $3T$ wide pulse, as shown in waveform **(a)** of figure 2.2. After propagation delay Δ, this signal arrives at the receiver. The received waveform is shown in **(b)**; instead of delaying the waveform, we have shifted the ticks on the time axis by Δ. As shown in **(b)**, the *strong* high-voltage pulses have expanded at the expense of the *weak* low-voltage pulses; also the signal is now asynchronous relative to the local clock. The little vertical arrows indicate the moments where sampling occurs. The samples seen by the receiver are shown in **(c)**. One can observe that the sampling outcome is 2 to 4 samples per bit, depending on the bit pulse width and on the point where the sampling occurs, e.g. for the ‘‘contracted’’ last 0-bit we can get 2 or 3 samples, depending on the "shifting" of the signal on the time axis. That is, the number of samples obtained for each bit is variable, and depends on the value of the pulse, *strong* or *weak*, and on the phase difference between the received signal and the sampling clock.

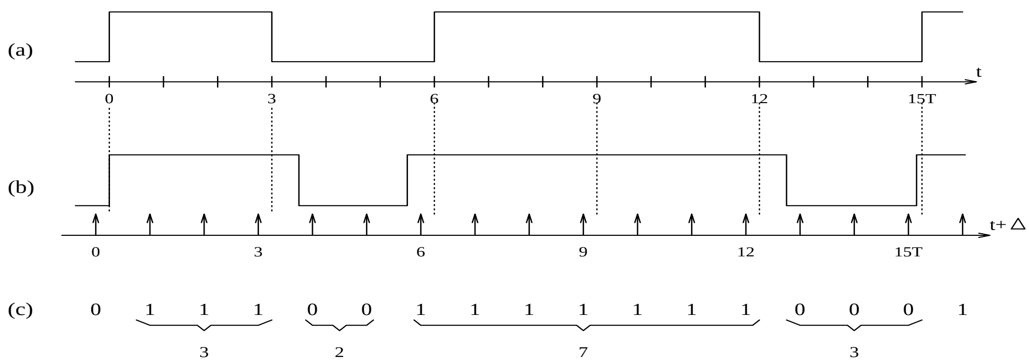


Figure 2.2: Example of signal distortion and sampling effect on pulse width ((N=3).

Part **(a)** shows the transmitted signal for "10110" and N=3, part **(b)** is the received signal (notice that the time axis was shifted by the transmission delay Δ) and sampling instants (asynchronous to transmitter and to waveform), and **(c)** shows the samples seen at the receiver.

2.2 Signal Preprocessing Filters

The Receiver handles the input signal distortion in two ways: using pulse preprocessing circuits, and a ‘‘smart’’ finite-state machine (FSM) that tolerates and correctly interprets numbers of samples that are not integer multiples of "*N*". The operation of the preprocessing circuits is outlined below, while the detailed circuits description is given in section 3.3. First, the input signal is preprocessed by a Variable Input Threshold Inverter. The Threshold Input Voltage of an inverter, V_{th}, is defined as the point where the output voltage equals the input voltage. By changing the value V_{th} we change the transition point of the inverter, thus for the same input pulse and for different values of V_{th} we get longer or shorter pulses at the output. Briefly, the operation of this circuit is the following: a user-settable mask provides the gate input to each of the transistors of the circuit; by setting certain bits in that mask, we selectively turn on and off transistors in the circuit; the equivalent circuit is an inverter with variable w/l ratio of the pmos and the nmos transistors; since the V_{th} depends on this ratio, we can change this voltage. The circuit consists of 3 pmos transistors and 3 nmos transistors. Each type of transistors is made with w/l ratios of 1, 2, and 4. The combinations of the transistors that are turned on result to the equivalent inverter circuit having ratio of the $\frac{(w/l)_p}{(w/l)_n} = \frac{A}{B}$, where $A, B = 1, 2, 3, .., 7$.

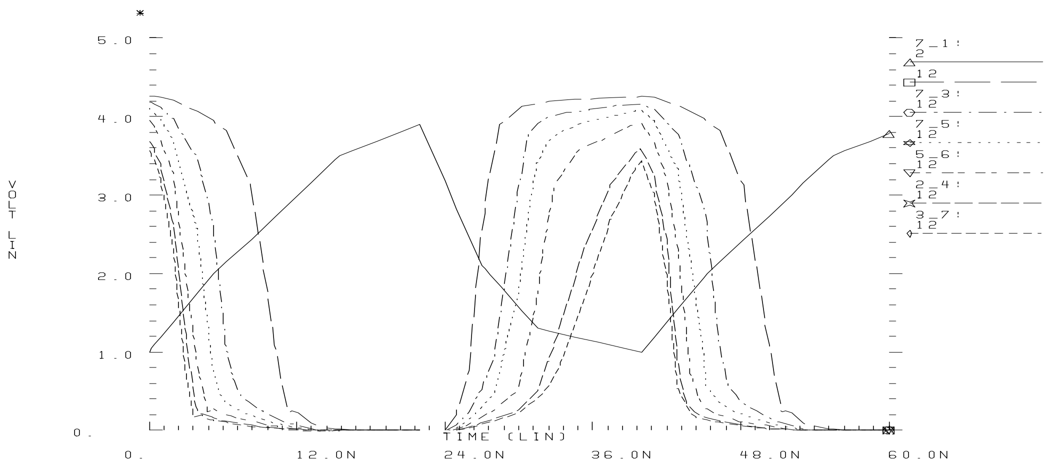


Figure 2.3: Simulation Results for the Variable Input Threshold circuit.

Figure 2.3 presents spice simulations of the Variable Threshold stage for different values of the pmos to nmos ratio. The input pulse, shown with the solid line, is representative of the expected received signal: it is similar to the output of the coaxial cable when we transmit successive high and low pulses at 50 MHz using our experimental setup. The resulting pulses are drawn with dashed lines, and, starting from the wider to the narrower, they correspond to pmos to nmos ratio of 7:1, 7:3, 7:5, 5:6, 2:4, and 3:7. We can see that the output pulse of the inverter can be modified significantly.

The second stage of preprocessing is a programmable leading edge delay circuit. By setting an 8-bit mask, we select to delay by 0 to 6 units the rising or the falling edge of pulses. The effect of this circuit is illustrated in figure 2.4. The input signal (the output of the Variable Threshold circuit) is shown in (a). The waveforms (b), (c) and (c) result from the input signal by delaying it by 1, 2, and 3 units, by passing it through multiplexors. The select mask controls the multiplexors that choose the delay amount, and finally the delayed signal gets ANDed or ORed with the original input. The result of this ANDing is the delay of the positive edges of pulses, while the ORing delays the negative edges of pulses. In this way, we enlarge one polarity of pulses at the expense of the durations of pulses of the opposite polarity. In figure 2.4(e) we see graphically how the leading edge of the negative pulse was delayed by 3 units by ORing the signal (d) with the input (a). The inverse effect of the AND gate is shown in part (f).

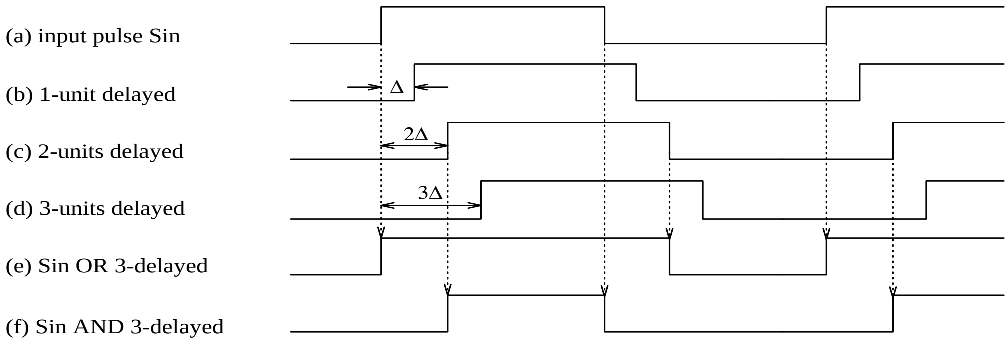


Figure 2.4: The effect of preprocessing of the Programmable Edge Delay circuit.

As we saw, using the two preprocessing circuits we can shape the input signal. The required transformation of the input signal depends on the bit time *N*, as will be analyzed in the "Decoding Algorithm" section. When *N* is odd we want the pulse shaping to restore strong and weak pulses to their original integer-*NT* width, while when *N* is even we want it to transform them to a width equal to $NT \pm \frac{T}{2}$. In a possible configuration, there could be an initialization phase of a communication where the preprocessing circuits are calibrated to counteract the signal distortion. During this phase, the preprocessing parameter values are set, and then the system is tested in order to measure the effect of these setting, then the parameters are corrected, and so on.

2.3 The Sampling and Decoding Algorithm

We have seen that the serial input stream is first preprocessed by the Variable Input Threshold and the Programmable Leading Edge delay filters. The aim of the preprocessing is to minimize the effects of signal distortion. After the preprocessing, the signal is sampled, and the resulting samples are fed to a finite-state-machine (FSM). This FSM decides what data bits are likely to have been transmitted, based on the number of identical samples seen in the serial input and on the value N of the bit length, according to the following algorithm:

Decoding Algorithm

Assume that we have received a continuous pulse of binary value V, of duration S samples. Find the unique integer number n such that:

if N is odd	if N is even	
$nN-\frac{N-1}{2} \le S \le nN+\frac{N-1}{2}$	$nN-\frac{N}{2} \le S < nN+\frac{N}{2}$	if V is the weak value
	$nN-\frac{N}{2} < S \le nN+\frac{N}{2}$	if V is the strong value

Then, decide that n data bits of value V have been received.

The general idea of this decoding is to tolerate a pulse distortion of up to $\pm\frac{1}{2} \cdot (bit-time)$. When N is odd (e.g. N=5), this is an unambiguous criterion: tolerate up to $\frac{N-1}{2}$ (e.g. 2) samples more or less than what the integer-NT pulse width would imply. When N is even, however, the question arises of what to decide when exactly $((n+\frac{1}{2}) \cdot N)$ (or equivalently $((n+1)-\frac{1}{2}) \cdot N$) samples have been seen. In that case, strong pulses are counted as n bits (because presumably they started as nNT-long, and they expanded to $(n+\varepsilon) \cdot NT$), while weak pulses are counted as $(n+1)$ bits (because presumably they started as $(n+1)NT$-long, and then shrank to $(n+1-\varepsilon) \cdot NT$).

2.4 Derivation of the Pulse Distortion Tolerance

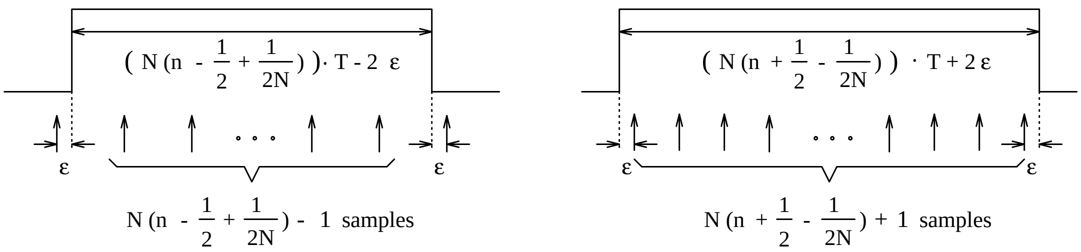
Based on the above algorithm and on the expected distortion by the physical medium, we can now compute the overall distortion that the FSM can tolerate. We have to distinguish between odd or even cases, as above.

Case I: N is odd

In this case, the receiver FSM interprets *strong* or *weak* pulses in the *same* way. Since the single bit width is NT, n bits of the same value are transmitted as a pulse nNT wide. On the other hand, the receiver interprets a pulse as n bits, *iff* it sees a number of samples in the range:

$$N \cdot (n - \frac{1}{2} + \frac{1}{2N}) \leq samples \leq N \cdot (n + \frac{1}{2} - \frac{1}{2N}).$$

We will compute the maximum pulse distortion that can be tolerated without resulting in an error in the receiver by computing equivalently the minimum shrinkage that can produce an error, as illustrated in figure 2.5(a). This is the case where the pulse width is $N \cdot (n - \frac{1}{2} + \frac{1}{2N}) \cdot T - 2\varepsilon$, where $\varepsilon \approx 0$, and two sampling points occur just outside the pulse. This results in receiving only $N \cdot (n - \frac{1}{2} + \frac{1}{2N}) - 1$ samples, which are incorrectly interpreted as $n - 1$ bits.



(a) Minimum shrinkage to produce an error

(b) Minimum expansion to produce an error

Figure 2.5: Minimum distortion that produces an error, for N odd.

In an analogous way, the minimum expansion that can produce an error is given by a pulse of width $N \cdot (n + \frac{1}{2} - \frac{1}{2N}) \cdot T + 2\varepsilon$ ($\varepsilon \approx 0$), and for the sampling instant alignment of figure 2.5(b). In this case, the distorted pulse results in $N \cdot (n - \frac{1}{2} + \frac{1}{2N}) + 1$ samples being received, which are incorrectly interpreted as $n + 1$ bits. Combining the above cases, we deduce that the minimum pulse width change that can produce an error (with "favorable" sampling instant alignment) is $\pm N \cdot (\frac{1}{2} - \frac{1}{2N}) \cdot T$, in either direction of distortion (shrinkage or expansion). As a result, the amount of distortion that the receiver can handle for correct reception is: $|pulse\ distortion| < \frac{NT}{2} \cdot (1 - \frac{1}{N})$. A point that must be emphasized is that in this case (N odd) pulse shaping should exactly equalize (eliminate) the average strong/weak distortion, since the FSM treats both types of pulses in the same way.

Case II: N is even

In this case, the FSM treats strong and weak pulses differently. Let us first study the case of

strong pulses. The transmitter transmits n *strong* bits as a pulse of duration nNT. The receiver interprets correctly a **strong** pulse as n bits *iff* it sees a number of samples in the range: $N \cdot (n - \frac{1}{2} + \frac{1}{N}) \le samples \le N \cdot (n + \frac{1}{2})$, according to the decoding algorithm. Following the same analysis as in case I, the minimum shrinkage that can produce an error is the one resulting in a pulse of duration $N(n - \frac{1}{2} + \frac{1}{N})T$, while the minimum expansion to produce an error is when a pulse of duration $N(n + \frac{1}{2})T$ results. We can now find the requirements for correct reception: If we define the change of the pulse duration as $\Delta t = Duration_{at\ receiver} - Duration_{at\ transmitter}$, then the rule for correct reception is: $-\frac{NT}{2} \cdot (1 - \frac{2}{N}) < \Delta t < \frac{NT}{2}$. In this case (N even), we arrange the pulse shaping (preprocessing) so that the transmission distortion **plus** the preprocessing of the **strong** pulse contribute $\Delta t_{pp} = T/2$. Now, the change of the pulse width, as seen by the receiver, is due to random noise and to the preprocessing, that is $\Delta t = \Delta t_{pp} + \Delta t_{randomnoise}$. From the above formulae, we get an expression for the noise: $\Delta t_{randomnoise} = \Delta t - \frac{T}{2}$ =>

$$-\frac{NT}{2} \cdot (1 - \frac{2}{N}) - \frac{T}{2} < \Delta t_{randomnoise} < \frac{NT}{2} - \frac{T}{2} \quad => \quad |\Delta t_{randomnoise}| < \frac{NT}{2} \cdot (1 - \frac{1}{N}).$$

For the weak pulses, we follow the same analysis as before, but here the preprocessing **plus** the transmission distortion contribute $\Delta t_{pp} = -T/2$- the opposite of the strong value preprocessing. For correct reception, the following equation holds: $-\frac{NT}{2} < \Delta t < \frac{NT}{2} \cdot (1 - \frac{2}{N})$.

Then we have:

$$\Delta t_{randomnoise} = \Delta t + \frac{T}{2} \quad => \quad -\frac{NT}{2} + \frac{T}{2} < \Delta t_{random\ noise} < \frac{NT}{2} \cdot (1 - \frac{2}{N}) + \frac{T}{2}$$

As a result we get $|\Delta t_{noise}| < \frac{NT}{2}(1 - \frac{1}{N})$, which is the same expression as the one for strong pulses, as well as for odd values of N.

We see that in all cases - odd or even N - the maximum tolerated width distortion for a pulse of n contiguous data bits of the same value is given by the same formula :

$$|pulse\ width\ distortion| < NT\ \frac{1 - \frac{1}{N}}{2} = \frac{1 - \frac{1}{N}}{2} \cdot (bit\ time) = \frac{bit\ time}{2} - \frac{T}{2}.$$

Table 2.2 shows numerical examples for the duration that can be tolerated, for various values of N, and for $T = \frac{1}{2f} = 8.33$ ns, for a clock of frequency $f = 60$ MHz:

As a further example let us examine the extreme and most interesting case where the bit duration is equal to the clock period, i.e. the bit length is $N=2$. In this case, we achieve the maximum rate of communication, by receiving based on 2 samples for each bit, sampling both on the

Table 2.2: Tolerable Distortion of the width of a contiguous-bit pulse (f=60 MHz, T=$\frac{1}{2f}$=8.33 ns, bit-time=NT)

N	bit-time ns	signaling rate Mb/s	Tolerated pulse width distortion as % of bit-time	Tolerated pulse width distortion in ns
2	16.7	60	25%	4.2
3	25	40	33%	8.3
4	33.3	30	37%	12.5
5	42	24	40%	16.7
6	50	20	42%	21
7	58	17.1	43%	25
8	67	15	44%	29
12	100	10	46%	46
24	200	5	48%	96
60	500	2	49%	246
120	1000	1	49.6%	496
1875	15.6 µs	64 Kb/s	50%	7.8 µs
6122	51 µs	19.6 Kb/s	50%	25.5 µs

positive and the negative edge of the clock. The decoding algorithm for N=2 is presented in figure 2.6. We can see that if the receiver sees 2 or 3 samples of the *strong* value it considers them as one bit, 4 and 5 samples as two bits, and so on. For the *weak* value, 1 or 2 samples are interpreted as 1 bit, 3 or 4 samples as 2 bits, and so on.

The behavior of the algorithm in this case, is based on the following observation. Provided that the pulse width after preprocessing is longer than 1 clock period, T, (the nominal width is 2T), there will be at least one sampling instant in the pulse, and we will get *at least one* sample of that bit.

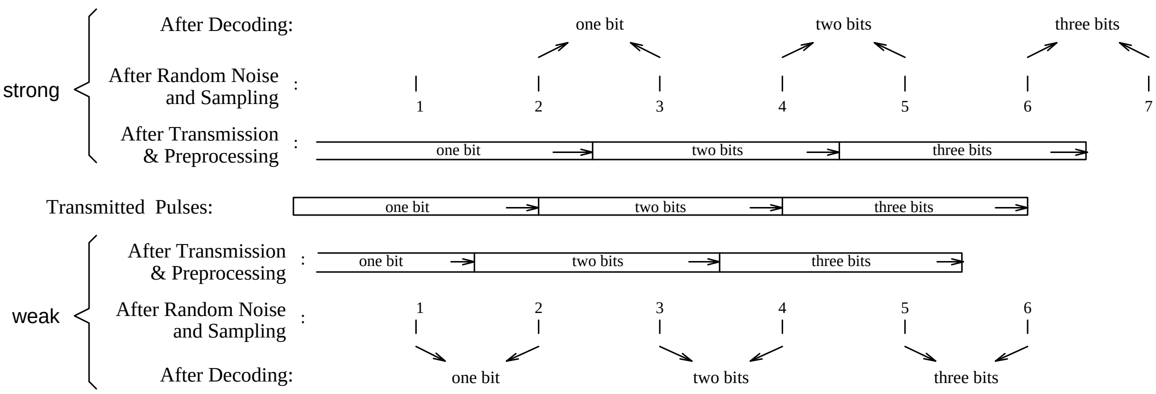


Figure 2.6: Samples distribution by the receiver FSM, for bit length = 2.

The overall system operation, for *N*=2, is exemplified in figure 2.7. We assume that the *strong* value is the 1, and that the bit sequence "101101" is transmitted. Part (a) shows the transmitter output, which is synchronous to the clock. Part (b) shows the raw and preprocessed input to the Receiver, as well as the sampling points. The raw input signal, drawn with the dashed line, is preprocessed through the pulse shaping circuit, and results to the signal drawn with the solid line. Part (c) shows the samples taken at the sampling instants (arrows), which are subsequently fed to the FSM. According to figure 2.6, the FSM will correctly recognize the sequence.

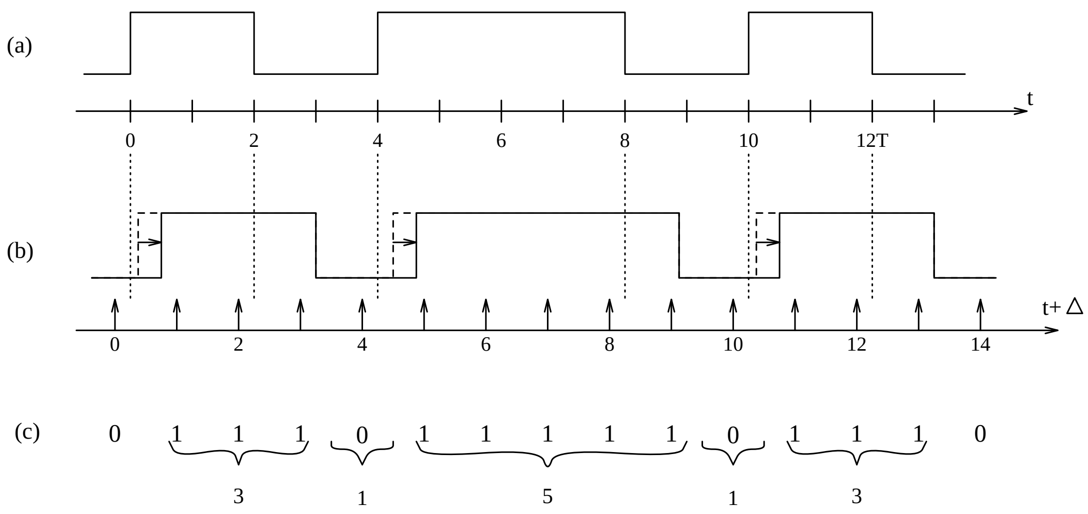


Figure 2.7: The effect of signal distortion and sampling on pulse width for N=2.

Chapter 3

Transmitter and Receiver Implementation

In this chapter we present the organization of the transmitter and the receiver. First, we present and evaluate the available standard cell library. Next, the transmitter circuits are discussed, followed by a presentation of the preprocessing and sampling circuits. Finally, we give an analysis of the receiver implementation, and the receiver's two-bit-per-clock Finite State Machine.

3.1 Available Cells

Our chip was designed using the 1.5 μm CMOS, double-metal standard cell process of ES2 (European Silicon Structures). The standard cell library of ES2 contains primitive cells, i.e. cells that are layout, and low to medium complexity macro cells, i.e. cells that are built up from primitive cells. More specificly, the primitive cell library contains all basic gates with up to three inputs, OAI and AOI structures, D flip-flops and latches with reset, clear, tristate drivers, 2 to 1 multiplexor, and strong inverting buffers. Some of the cells are also available as quadruples, i.e. four 2-1 multiplexors with common select, and a four-bit register. Gates with greater than 3 fan-in exist as macrocells. The macrocell library contains equivalents of popular parts from the 74LS library, several types of flip-flops (JK, T, SR latches), gates with large drive strengths, shift registers, counters, multiplexors and, demultiplexors. Large macrocells, that is PLAs, ROMs, RAMs, and multipliers are generated using predesigned layout cells. However, there is no generator for building variable size adders.

Some comments on the primitive cells of the library are that:

1) The commonly used NOR gates have low drive strengths and are slow compared to the equivalent NAND gates, for the same load. For example, the 3-input NOR has a maximum

propagation delay of 3.9 ns, while this delay is 2.3 ns for the 3-input OR, and 1.5 ns for the 3-input NAND gate! Thus, one must be aware of the characteristics of the library before designing. ‘‘Blind’’ selection through a menu may result to large and slow circuits.

2) All types of flip-flops have both the primary output as well as the complementary output. However, only in very few cases are both of these outputs used, and even in these cases the user could just generate the second of them with an inverter. A flip-flop with a single output would be more compact and thus a better cell for the library.

3) The PLA layout synthesizer generates very slow PLAs, compared to the equivalent logic built up from basic cells.

4) The library includes only edge-triggered flip-flops, but not dynamic latches. Dynamic latches do not usually appear in standard cell libraries; however, they are much smaller than edge triggered flip-flops and their use would significantly reduce the occupied area, and they also give more flexibility and control over the design.

Throughout our design, we used mainly the basic cells and some of the lower level macrocells. This type of design was preferred over designing with macro-cells, because it gives an understanding of the lower structures and better control over the expected result. Also, it is necessary if high-speed operation of the circuit is desired, since the standard cell library is not always the optimum design in terms of speed. In one case, full-custom layout was necessary, for the programmable threshold voltage stage; it was designed at the lowest level, according to the design rules of ES2.

3.2 The Transmitter Part

"Half" of the UART cell (the easier half) is the Transmitter. It works independently from the Receiver. The Transmitter accepts parallel data from an external interface, converts them to a serial bit stream, inserts the appropriate start, stop and optional parity bits, and outputs the composite serial data on the *SerialOut* port. Its operation is programmable through the setting of parameters for the parity, character length, and baud rate. The rate of transmission can be any integer submultiple of twice the clock frequency, up to f_{clk}. This is achieved by being able to switch the output value on either the rising or the falling edge of the clock. In the next paragraphs we describe the interface to the external circuits and the implementation of the Transmitter.

3.2.1 Interface to external circuits.

The Transmitter communicates with the external circuits via two groups of signals: the configuration setting signals, and the data and status output signals. The configuration signals include input parameters and load enable signals. For each parameter there is a separate load signal, so that it can be configured independently from other parameters. We preferred having each parameter loaded separately, instead of loading all parameters in parallel, because the former fits better to the case of loading values through a bus. The parameters are Xbitlen, Xchlen, and Xparity, and the corresponding load enable signals are Xld_bitlen, Xld_chlen, and Xld_par, as shown in figure 3.1.

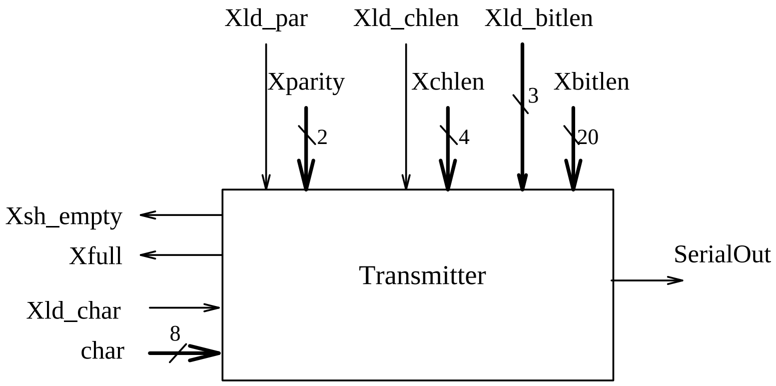


Figure 3.1: Transmitter interface signals.

The other group of signals consists of data, request-acknowledge, and status signals. The *Xsh_empty* and *Xfull* signals are status signals, produced by the transmitter's control logic. As illustrated in the transmitter's block diagram, in figure 3.3, external data are double buffered; first they are loaded in the *hold* register, and then they are moved to the *shift* register and get transmitted, as soon as this becomes empty. The *Xfull* signal denotes that the *hold* register contains data, while the *Xsh_empty* signal denotes that the *shifter* is empty of data. The negative of the *Xfull* signal can be used as a "request for data". The external circuit responds to this request for data, supplying the next data byte on the *char* input lines, and asserting the *Xld_char* signal. The timing of these signals is illustrated in figure 3.2, for bit length N=3. Input data are latched on a negative clock edge, when *Xld_char* is asserted. The *Xsh_empty* signal is synchronous to the transmit clock *xclk*; it is reset when the *start_bit* of a character is being transmitted, and it is set at the end of the transmission, when the *stop bit* is being transmitted. The *Xfull* signal is asserted when new data are loaded in the *hold register*, and it is deasserted on the first rising edge of *clk* after the start of transmission.

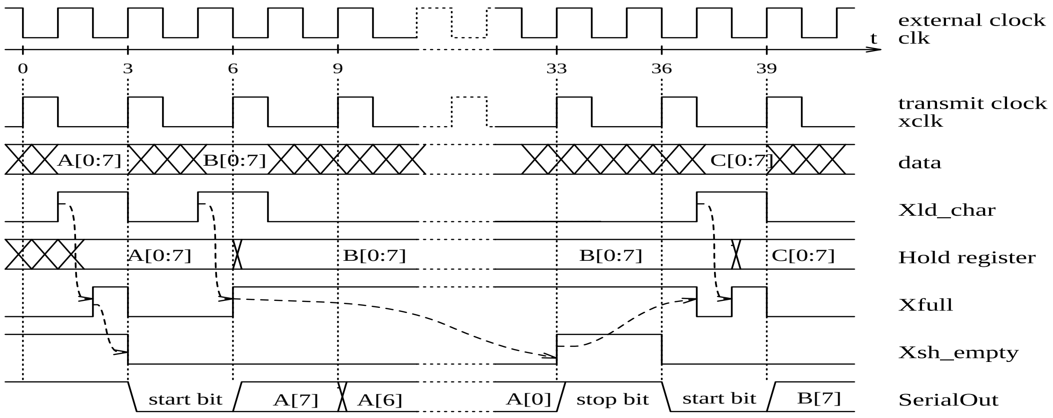


Figure 3.2: Transmitter interface timing, for bit length N=3.

The external clock source *clk* is divided by $3N/2$ and the resulting clock *xclk* is used as the transmit clock. Initially, at t=0, the transmitter is in the idle state, where it has nothing to transmit; in this case, the *Xfull* signal is low, and the *Xsh_empty* signal is high. We assume that at t=1, the external circuit asserts the *Xld_char* signal in order to give the 8-bit character *A* for transmission. Input data are latched in the *hold* register on the negative *clk* qualified with the negative *Xld_char* signal. The *Xfull* signal is asserted synchronously to the data loading, and in the following *xclk* cycle (t=3) the *Xsh_empty* signal goes low, as the transmission of *A* begins.

As mentioned in the above paragraph, the *Xsh_empty* signal goes high before the end of character transmission, when the stop bit is being transmitted. This signal timing is one cycle "early", in order to let us prepare for a new transmission, if another character is ready in the *hold* register. If so, the transmission of the new character will start immediately after the end of the current one, with no intermediate delay. That is, the transmitter works in a continuous, pipelined-way; this feature is especially useful for transmitting data bursts. In order to have two characters transmitted "back-to-back", the external circuit must load the second character in any of the intermediate cycles between the deassertion of *Xfull* and the assertion of *Xsh_empty*. The latest time that a character can be loaded and be transmitted with no intercharacter delay, is on the *clk* cycle(s) after the assertion of *Xsh_empty*. If the external circuit forces a character loading when the *Xfull* signal is high, data in the *hold* register will be overwritten.

3.2.2 Transmitter Circuit Description

A general block diagram of the Transmitter is shown in figure 3.3, while the detailed schematics are given in the Appendix A. The Transmitter contains 5 blocks:

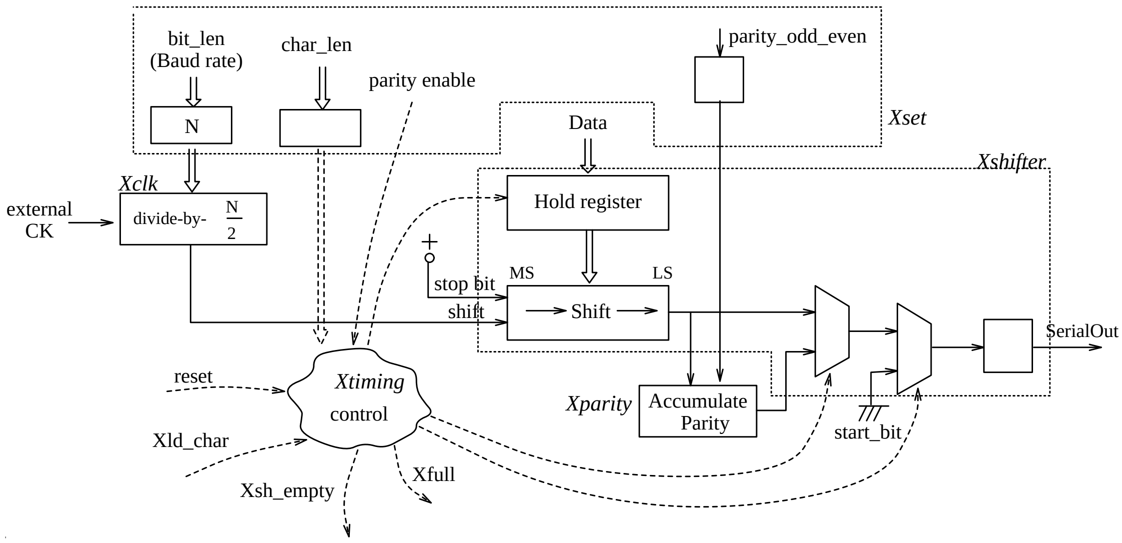


Figure 3.3: Transmitter block diagram.

Xset

The Xset block holds the parameters for the Transmitter operation, i.e. parity, character length, and bit time duration. Specificly, it contains a 2-bit register for *Xparity_enable* and *Xparity_odd_even*, a 4-bit register for character length, *Xchlen*, that can take one of the values 5,6,7 or 8, and a 20-bit register for the bit length *Xbitlen*, expressed as a multiple of the external clock *clk* half-period. The external circuit can load each register by setting the appropriate load signal *Xld* for each parameter. Loading through an 8-bit bus is supported, and thus, for parameters wider than 8 bits, e.g. for the 20-bit *Xbitlen*, multiple load signals are provided, e.g. *Xld_bitlen* [0:2]. Moreover, the data pins are available all in parallel, and the connections to the bus are left to be made outside the macrocell. This makes the macrocell suitable for configurations where a wider bus (e.g. a 32-bit bus) is present; in this case, several load signals will be tied together.

Xclk

The *Xclk* block generates the transmit clock *xclk*, whose period is equal to the bit time. The *xclk* is produced by dividing the external clock *clk* by *Xbitlen*, which is variable and ranges between 2 and $2^{20}-1$. The transmit clock period, and thus the bit interval, is given by $Xbitlen * \frac{T_{ext}}{2}$, where T_{ext} is the period of the external clock *clk*. For example, for an external clock of 25 MHz, the transmit rate can vary between 25 Mbaud and 47 baud.

The division of the clock frequency could be made using a binary counter that would count on both the rising and the falling edge of the clock. When reaching a count of *Xbitlen*, it would produce an *xclk* pulse. However, such a circuit is not available in the standard cell libraries, so it has been synthesized from other cells. Its main part is a 19-bit down-counter, with carry look-ahead, that counts on the rising edge of clock, and that is loaded asynchronously with the value $\frac{Xbitlen}{2}$. The desired counter is simulated by appropriately loading this counter: it is loaded by the transmit clock, *xclk*, rather than by the external clock, *clk*. Other parts of the circuit are two flip-flops, the *cnt_unit* and *cnt_zero,* that keep the information whether the current counter value is "1" and "0". These flip-flops are negative-clock-edge triggered, thus they track the counter value shifted by half *clk* cycle. To describe the operation of this circuit, we have to distinguish between two cases:

1) if the bit length is even (the easy case): after the counter counts $\frac{Xbitlen}{2}$ periods (which is an integer), it is asynchronously loaded. The load condition is expressed as: *cnt_unit*clk*, which is the same as loading the counter when it is zeroed. This case is depicted in figure 3.4(a):

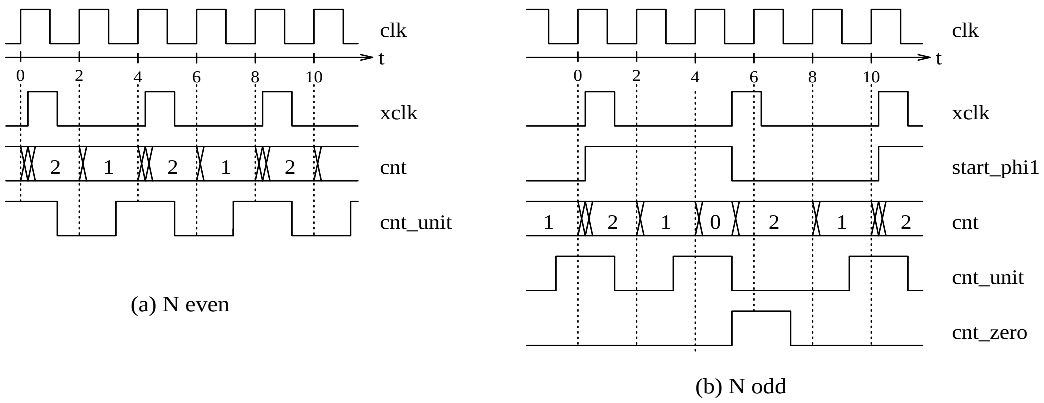


Figure 3.4: Timing waveforms of Xclk block for bit length (a) even (N=4), and (b) odd (N=5).

The *xclk* signal is the transmit clock, generated by dividing the external clock source *clk* by $N/2$. The contents of the 19-bit counter are depicted by the *cnt* signal, and the *cnt_unit* and *cnt_zero* are the omonymous flip-flop contents.

2) if the bit length is odd, then we have to count N half *clk* periods, or equivalently $(N-1)/2$ *clk* full periods and one *clk* half period. Let us assume that initially, we load the counter on the rising edge of the clock with the value $(N-1)/2$, that is with the 19 MS bits of the bit length value. This occurs at t=0 in the example of figure 3.4(b), and for $N=5$. After $(N-1)/2$ cycles (at t=4 in the figure), the counter reaches zero. Counting one half-period remains to complete the counting cycle; this is accomplished by reloading the counter when the *clk* goes low. The next counting cycle begins at t=5, and at t=6 we have already

counted the half-period and what remains is counting $(N-1)/2$ full periods. We keep the information of whether counting begins on the high *clk* half-period or on the low one in the *start_phi* 1 latch. The above conditions for the counter loading are expressed by:

$$(cnt\_zero * start\_phi\,1) + (cnt\_unit\ *\ !start\_phi\,1\ *\ clk).$$

Xtiming

The Xtiming block implements the FSM of the Transmitter states, and produces the timing signals indicating the start of a transmission, the parity bit time, and the end of transmission. The circuit's operation can be represented by the two FSMs of figure 3.5. The first FSM generates the *Xfull* signal, according to the state of the *hold* register, while the second represents the transmit cycle. The communication point between the FSMs is the *init_x* signal. The *init_x* signal initializes a new byte transmission, and is produced when the shifter is idle and a new character is available in the *hold* register; it is given by the expression *Xsh_empty*Xfull*. Parity bit transmission is enabled by the *xmit_parity* signal, produced in the *Xparity* state of the FSM. Finally, the *eocx* signal, produced in the *EOCX* state, means the end of a character transmission. The transmit FSM is implemented by a decrementor, which is loaded with the character length on *init_x*, and counts down with the rising edge of *xclk*.

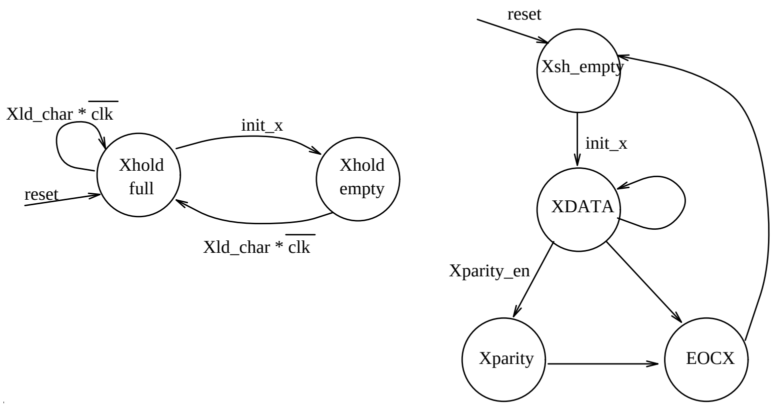


Figure 3.5: Timing Finite State Machine.

Xparity

The *Xparity* block generates the parity bit, to be transmitted if parity is enabled (Xparity_enable HIGH). It is implemented as a sequential circuit with a D flip-flop and 4 logic gates, rather than as an 8-input combinatorial (parallel) parity generator. At the initialization phase of a character transmission (*init_x* HIGH), the flip-flop is loaded with 0 or 1, if parity is even or odd, respectively. After that, the flip-flop is repeatedly reloaded with its contents XOR-ed with the bit currently transmitted. In this way, in case of even parity, the bits of the character are xor-ed together (the initially loaded 0 does not affect the result). In the case of odd parity, the XOR'ing of the character bits is inverted by the initial 1, which is equivalent to an NXOR gate producing odd parity.

Xshifter

The *Xshifter* block converts the parallel input data into serial, as well as inserting the start bit, the stop bit, and optionally the parity bit. It is implemented as a 9-bit shift register, with the last stage modified for parity bit insertion, as illustrated in figure 3.6. It is loaded synchronously, by a *init_x* pulse, when a new transmission begins.

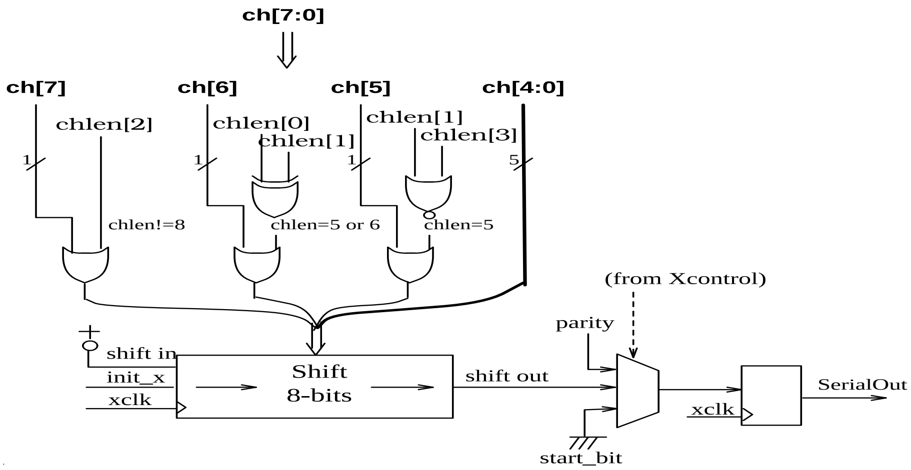


Figure 3.6: Transmitter Shifter circuit.

The serial input of the shifter is hardwired to the supply, to provide a logic 1 for the stop bit(s) of a character. The parallel input of the shifter consists of the variable number of character bits, padded with 1's to 8 bits, if required (that is for character lengths 5,6 or 7). This padding is done by OR'ing the data bit with one condition. If the condition holds, then the bit is padded with "1". For the data bit *i* (the LS bit is 0), the condition is "*Xchlen*>*i*", and is easily evaluated from specific bits of the *Xchlen* value, as shown in figure 3.6. The least significant bit is rightmost and it is the first one to be shifted out. The stop bit does not hold a stage in the register, but is rather shifted in by the serial input sin of the register. Thus, while the character is shifted out, the register is filled with 1's from left to right, and, at the end of the transmission, its output will remain to logic HIGH, which is the idle state. Upon RESET, the shifter is automatically initialized to 111..1, in 9 xclk cycles, using the serial_in input.

The last output stage of the register includes a mux for inserting the start bit and the parity bit. The parity bit is calculated during character bit transmission, selected by the *xmit_parity* signal and loaded in the last D flip-flop. The mux is placed before the FF and not after it, so that the duration of all output pulses is constant, generated always by the xclk clocking this last flip-flop.

3.3 The Preprocessing and Sampling circuits

The receiver is able to tolerate a considerable amount of distortion. This is accomplished first by preprocessing the input pulses, and secondly by a smart FSM. In this section, we discuss in detail the preprocessing and the sampling circuits. The first stage of input signal processing consists of a variable threshold voltage circuit. It is essentially an inverter, acting as an amplifier, with programmable ($\frac{w}{l}$) ratios of the pull-up and the pull-down transistors.

The Input Threshold Voltage V_{th} is a characteristic point of the transfer function of the CMOS inverter, where the input voltage equals the output voltage ($V_{in}=V_{out}$) [Uyem88], i.e. a point "at the middle" of the narrow range of input voltages across which switching of the output between the high and the low state occurs. At this point of operation, both the nmos and the pmos transistors are in saturation. Then, by equating the currents in the two transistors we have:

$$k_P \cdot (V_{dd}-V_{in}-|V_{TP}|)^2 = k_N \cdot (V_{in}-V_{TN})^2 \;\; => \;\; \cdots \;\; => \;\; \sqrt{\frac{k_P}{k_N}} = \frac{V_{in}-V_{TN}}{(V_{DD}-V_{in}-|V_{TP}|}$$

Since $V_{in}=V_{out}=V_{th}$, the above equation results in the following expression for the input voltage:

$$V_{in} = \frac{V_{DD}-|V_{TP}|\sqrt{\frac{k_P}{k_N}}+V_{TN}}{1+\sqrt{\frac{k_P}{k_N}}}.$$

Substituting the values of $V_{TP=-1.1V}$ and $V_{TN=0.7V}$ for the ES2 1.5 μm typical process, we get:

$$V_{in}=\frac{3.9\beta + 0.7}{1+\beta}, \text{ where } \beta=\sqrt{\frac{k_P}{k_N}} = \sqrt{(w/l)_p/(w/l)_n}\,.$$

From the above expression we can see that the input threshold voltage depends on the ratio of the (w/l) of the pmos and the nmos transistors. By changing this ratio we can change the V_{th} and thus, we can set the time during the (slow) input transition when the amplifier inverter switches.

The variable V_{th} circuit is shown in figure 3.6. It consists of 3 pairs of pmos (M1-M2, M3-M4, M5-M6), and 3 pairs of nmos transistors (M7-M8,M9-M10,M11-M12). Within each pair,e.g. in the M1-M2 pair, one of the transistors (M2) serves as a switch that connects or disconnects the other transistor (M1) in the circuit. In this way, by setting the select signals, sp0,sp1,..sn0, sn1,.., we can selectively cut off or use certain branches. The transistors with common gate input are equivalent to an inverter whose pmos (w/l) equals the sum of the w/l ratios of the participating pmos branches, and whose nmos transistor has (w/l) equals the sum of the w/l ratios of the participating nmos branches. By drawing the transistors ratioed in powers of 2, i.e. 1,2,4, we can synthesize an inverter with w/l of the form $\frac{A}{B}$ where $A,B=1,2,3,..,7$. For example, to have effective w/l of the equivalent inverter equal to 3/7, we must set sn1,sn2,sn4 to 5V and sp5 to 0V.

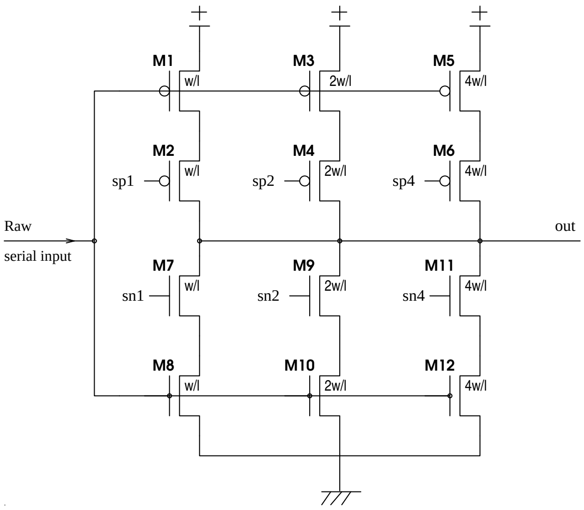


Figure 3.7: Variable Input Threshold Stage.

This cell has been laid using the design rules of the ES2 1.5 μm technology. The resulting cell dimensions are 45 x 60 microns. Each enable transistor in a pair is matched to the other transistor, so that it does not affect the above calculations. A comment should be made on the position of the transistors within a pair. Since the serial input is a signal with fast changes, the transistors whose gates are connected to this signal switch very often between conductance and cut-off. On the contrary, the enable transistors are usually in the same state, because the select signals are set once in the beginning and remain stable afterwards. In our design, we put the enable transistors on the side of the output. The benefit of this configuration is that: (1) it minimizes the Miller-effect capacitance on the input line due to non-participating branches, and (2) it minimizes the capacitance on the output line due to the same branches.

The next step of pulse preprocessing is the programmable edge delay (for either the positive or the negative edges), which is done by the *Rmod_pulse* circuit, as shown in figure 3.8. This circuit receives its input, min, from the *Rvthresh* unit, and its output, *mout*, feeds the synchronization circuit *Rsync*. It consists of a chain of multiplexors, $m7$ to $m1$, each of which selects an input either from the serial input or from its previous multiplexor output in the chain, according to the $modmask_i$ select signal.

If we define as 1-unit delay the delay introduced to the original input by one stage of this circuit (one mux), the total delay of the original input signal through the chain is programmable between 0 to 6 (+ 2 inverter delay) units. The delayed signal, *md*, is ANDed and ORed with the original signal. The AND gate has the effect of delaying the positive edges by the amount of

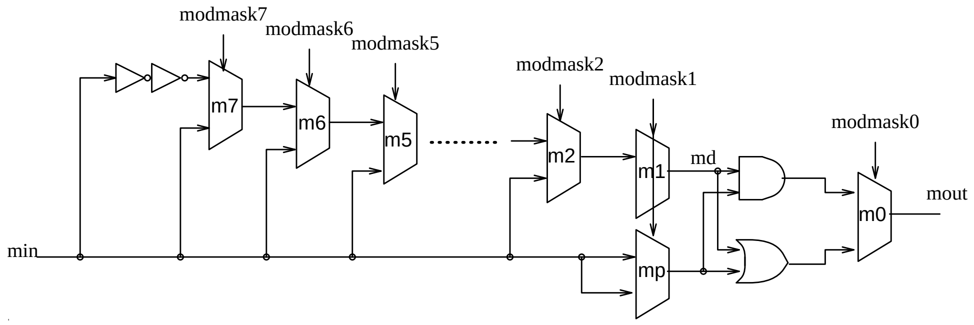


Figure 3.8: Receiver Programmable Edge Delay circuit.

delay that the *md* signal bears. The OR gate delays the negative edges by the same amount. The last multiplexor, *m* 0, selects which of the two types of edges will be delayed. In some cases, no edge delay may be desired. To enable this, we use the *mp* multiplexor that is placed in parallel to m1, and "equalizes" the delay of that multiplexor.

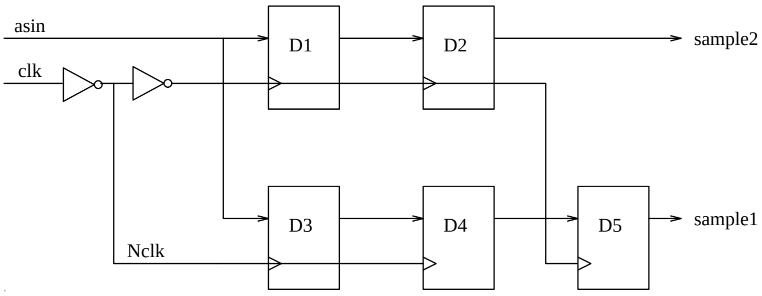


Figure 3.9: Rsync Synchronizing circuit.

The next step is the sampling of the input signal, and the synchronization of the samples to the receiver clock. The receiver's clock runs at approximately the same average frequency as that of the transmitter, but not exactly the same; this is a ''plesiochronous'' synchronization scheme, as defined in [Mess90]. The main problem that is inherent to the plesiochronous type of interconnect is the metastable behavior. Metastability is defined as the anomalous behavior of all bistable devices, in which the device gets stuck in an unstable equilibrium between the two states for an indeterminate period of time. In our case, it occurs because a signal that was generated as synchronous to the transmitter clock, is sampled using the receiver's clock. The signal can change at any time, and its sampling can occur during a transition, resulting in a metastable flip-flop state. In order to keep the probability of metastable condition acceptably low, we pass the signal through a chain of flip-flops, [Mess90]. (Note: these flip-flops are *not* of the dynamic CMOS type; they are made of cross-coupled devices with positive feedback, which is necessary in order for this circuit to switch away from the metastable state).

Another feature of the *Rsync* synchronizer is that is collects 2 samples per cycle, sampling on both the rising and the falling edge of the clock. The upper chain of flip-flops, D1 and D2, sample on the rising edge, while D3 and D4 produce a sample on the falling edge of the clock. The D5 flip-flop delays the output of D4 by an extra half cycle, so that both samples be available to other circuits on the rising edge of *clk*.

3.4 The Receiver Circuits

Apart from the transmitter, the other "half" of a UART cell is the Receiver. This unit takes input from the serial stream "Serin", removes the start and stop bits, and converts the rest into byte words, detecting at the same time possible transmission errors. The Receiver works in a configurable manner, where parameters can be set by the external circuit. Besides the characteristics of a conventional UART receiver, it has a wide working range; the maximum receive rate equals the frequency of the external clock source, while the minimum receive rate is the quotient of the external clock source divided by $2^{20}-1$.

3.4.1 Interface to external circuits.

The external interface of the receiver is illustrated in figure 3.10.

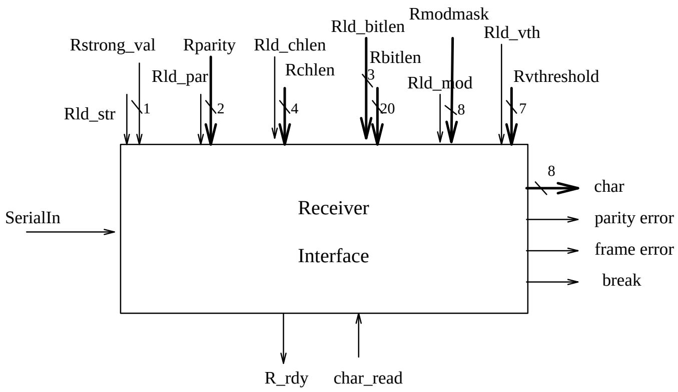


Figure 3.10: Receiver block diagram with interface signals.

On the left side of the block, are shown the configuration parameters and their load signals. The value and the load-enable wires are separate for each parameter, so that the user of this receiver cell has the freedom to load them either one-by-one through a narrow bus, or all in parallel from a wide connection. The narrowest such bus assumed is 8-bit wide. Thus, for parameters that are wider than one byte, multiple load signals are provided. For example, the load enable *Rld_bitlen* that corresponds to the 20-bit *Rbitlen* parameter, consists of 3 signals, each of them enabling the MS byte, the next MS byte and the LS 4 bits of the input value. The configuration parameters are the *Rvthresh*, *Rmodmask*, *Rbitlen*, *Rchlen*, *Rparity*, and *Rstrongval*, while the corresponding enable signals are the *Rld_vth*, *Rld_mod*, *Rld_bitlen*, *Rld_chlen*, *Rldpar*, and *Rld_str*.

The data output of the receiver is the *char* output which contains the character last received. If the character length is less than 8 bits, the MS unused bits are set to 0. Together with the *char* register, 3 bits of status information report whether a parity error, a frame error, or a break condition were detected during reception. The *R_rdy* signal is a notification signal to the external circuits, and is asserted when a new, valid character has been received and appeared on the char output. More specificly, it is asserted on the falling edge of the recovered clock *rclk* when the stop bit of a character is being received. The *char_read* input is the acknowledge of the external circuit that the received character has been read. The *char_read* also clears the *R_rdy* output (see § 3.4.2 Rstatus, for details on the timing of this signal).

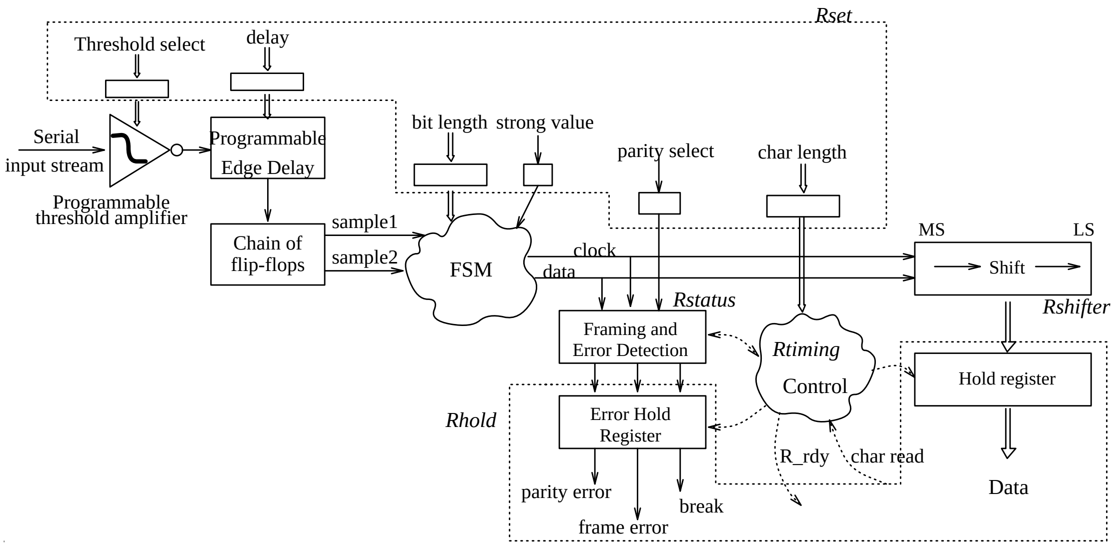


Figure 3.11: Receiver block diagram.

3.4.2 Receiver Circuit Description

A general block diagram of the Receiver is shown in figure 3.11, while the detailed schematics are given in the Appendix A. The Receiver contains 5 blocks:

Rset

The *Rset* block holds the settable parameters for the Receiver operation, that is:

> *Rvthreshold*, a 7-bit parameter that is used by the *Rvthrsh* preprocessing circuit. The 3 MS bits select the N-type transistors with increasing w/l, the next 3 bits select the P-type transistors, and the LS bits selects to bypass or not the *Rvthresh* circuit.

Rmodmask, the 8-bit mask for the *Rmodpulse* programmable edge delay circuit. The 7 LS bits of modmask, denoted modmask[6:0], are used to select the delay amount, which can take values between 0 and 6 as shown in Table 3.1.

Table 3.1: Receiver Programmable Edge Delay amount specification

		widen positive pulses	widen negative pulses
widen by	0 units	1 1000000	0 1000000
	1	1 0100000	0 0100000
	2	1 0010000	0 0010000
	3	1 0001000	0 0001000
	4	1 0000100	0 0000100
	5	1 0000010	0 0000010
	6	1 0000001	0 0000001

Rbitlen is a 20-bit parameter for the bit length, in units of the external clock *clk* half-periods.

Rchlen for the character length. This can be one of 5,6,7 or 8 bits.

Rparity for the selected parity. It consists of 2 bits, the MS indicating if parity is enabled, and the LS indicating the parity type, i.e. odd or even.

Rstrongval, is a 1 bit parameter used by the Finite State Machine. It denotes the binary value that was favored by the transmission medium. That is, if the 1-pulses expand during transmission this value must be set to 1.

Rshifter

The *Rshifter* block, presented in figure 3.12, converts the serial recovered bit stream into parallel. If the bit length is less than 8 bits, it inserts 0s in the unused MS bits. It is implemented by a 5-bit shifter and 3 separate D-type flip-flops. In front of each flip-flop, a multiplexor selects between the output of the previous flip-flop, and the serial input. In this way, we build a variable length shifter, where the serial input selectively bypasses some of the first stages. The flip-flops that are bypassed are cleared. The shifter continuously shifts its contents, every *rclk* cycles. It is in the responsibility of the control logic to generate the signals that latch the shifter's parallel output into the holding register.

Rtiming, Rstatus, Rhold

The Rtiming block generates the timing signals used in the *Rstatus* and *Rhold* blocks. Upon reset, the FSM of this block (figure 3.13) goes to the initial wait state; this is also the state that it stays in during intercharacter idle time, where continuous 1's are received. When the first 0 (start bit) is received, the receive cycle begins and the FSM moves to the *Rstart* state. In the following cycles, the FSM moves successively through the Rdata states, counting the data bits being received, starting from the LS bit. The Rdata cycles are as many as stated by the character length

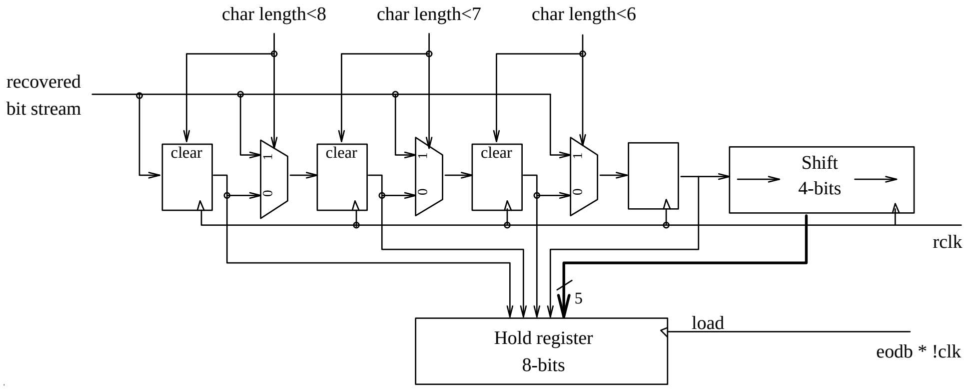


Figure 3.12: Receiver shifter circuit.

parameter. In the last data cycle, RdataMS, the *eodb* (End of Data Bits) timing signal is generated. The following state is the Rparity, for the parity bit reception, or the Rstop, for the stop bit reception, depending on the parity enable parameter. In the final state, *Rstop*, the *check_parity* and *eodb* signals are generated -the *check_parity* is generated only if parity is enabled-. In the next paragraph we will see where these signals are used.

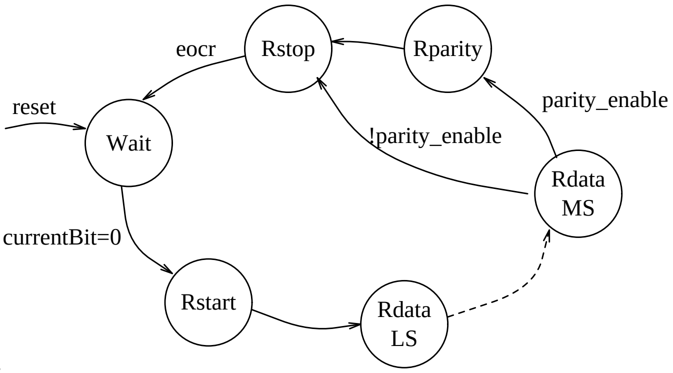


Figure 3.13: Receiver Timing finite state machine.

The *Rstatus* block detects some of the possible transmission errors. These errors include parity errors, frame errors and break condition. We have a parity error when the received parity bit is not the same as the one computed according to the received data and the parity type. The frame error is detected when a zero value is received at the time when the stop bit (high value) was expected. A break condition is detected when all bits of a character are 0, including the parity and stop bit(s).

The parity bit is computed using a parity accumulator, which is similar to the equivalent circuit of the transmitter. At the parity bit time, the current contents of the accumulator are XOR'ed with the currently received bit. If the result of this operation is 1 then we have a parity error, and in the last cycle, or equivalently in the Rstop state of the FSM, this result will be loaded in a latch by the *check_parity* enable signal. The frame error detection is straightforward: we set a latch with the current bit value, using as load enable the *eocr* signal. Finally, the break condition is detected using a similar accumulator as for the parity, except that in this case the contents of the accumulator are OR'ed with the current bit value. The result of this operation is also latched when the *eorc* signal is asserted.

Finally the *Rhold* circuit implements the double buffering. It includes an 8 bit register for the data, loaded from the shifter as soon as all data bits are received, using the *eodb* as load enable signal. The error bits are also loaded in a 3-bit register clocked by the *eocr* signal qualified with the negative transmit clock *rclk*. At the same time, the *R_rdy* signal is asserted. The external circuits, can read the value and clear the *R_rdy* latch by asserting the *read_char* signal. In case the received character is not read until the next character is received, it is overwritten.

3.5 The Receiver Bit FSM

The Receiver Bit FSM processes 2 samples at a time. Its inputs are the nominal bit length N, and the two samples acquired in each clock period, while its outputs are the value of a received bit, and a receive clock pulse, *rclk*, synchronous to the received data. Before describing the operation of the implemented FSM, we will first examine the operation of an equivalent FSM with 1 sample per clock period, since it is simpler.

3.5.1 The Receiver Bit FSM operating on 1 sample per clock period.

Table 3.2 presents the operation of this simplified FSM that processes 1 sample per clock cycle. The *input* row contains the samples received in successive clock cycles, while the next row shows the number of received samples for a specific bit value. A double vertical line draws the limits between bits. The start of a new bit is signaled either by a transition of the input, *ch* ("change"), (i.e. at clock cycle 1 we start receiving a bit of value S instead of the previous bit $\bar{S}$), or when the nominal number of samples for a bit has been seen; this is the case exemplified at clock cycle N, where all N samples of a bit of value S have been received, and thus, a new bit starts in the next cycle.

The FSM operation is based on a 20-bit decrementor, named *cnt* (counter), which is used to track the number of samples seen up to now for the current bit value. When the FSM sees a change of the input pulse, this decrementor is synchronously loaded with the value of the bit length N minus 1. Thus, in the next cycle the decrementor 's value will be $N-1$, indicating that

Table 3.2: Operation of the Receiver’s Finite State Machine with 1 sample per clock cycle

time (in clk cycles)	0	1	2	3		K	K+1		N-1	N	N+1	N+2		N+K+1		2N-1	2N	2N+1	2N+2		2N+K +1	2N+K +2
input	$\overline{S}$	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	$\overline{S}$	$\overline{S}$
# samples		1	2	3	...	K	K+1	...	N-1	N	1	2	...	K+1	...	N-1	N	1	2	...	1	2
cnt			N-1	N-2	...	N+1 -K	N-K	...	2	1	0	N-1	...	N-K	...	2	1	0	N-1	...	N-K	N-1
asserted signals		ch					**rclk**							**rclk**							**rclk** ch	

K samples = half bit

"$N-1$ samples remain to be received for the current bit". As soon as we have received the minimum number of samples for one bit, "K", and the decrementor has reached a count of $N-K$, a pulse of the receive clock *rclk* is output together with the value of the received bit. In the following paragraphs, we will refer to the count of the decrementor that produces an rclk pulse as cnt_{rclk}. When the decrementor reaches zero, then the nominal number of samples has been received and the reception of a new bit is initiated. In table 3.2, at clock cycle $2N+K+1$, we see the limit case where the decrementor has reached a count of $N-K$, and at the same time we see a transition of the input. In this case, the receiver FSM outputs an *rclk* pulse, since the *cnt* value of $N-K$ means that K samples have been received in the previous K cycles, and it initializes the reception of a bit of value $\overline{S}$.

Below, we will calculate the value of cnt_{rclk}, that is the condition to decide that "one more bit has been received". Remember from section 2.3, that the minimum number of samples that constitute a bit is K, where

$$K = \begin{cases} \dfrac{N+1}{2}, & \text{if } N = odd \\ \dfrac{N}{2}, & \text{if } N = even \text{ and } S = weak \\ \dfrac{N+2}{2}, & \text{if } N = even \text{ and } S = strong \end{cases}$$

Since $cnt_{rclk} = N-K$, we can find the equivalent expression for cnt_{rclk}:

$$cnt_{rclk} = \begin{cases} N - \dfrac{N+1}{2} = \dfrac{N-1}{2} = \left\lfloor \dfrac{N}{2} \right\rfloor, & \text{if } N = odd \quad (i) \\ N - \dfrac{N}{2} = \dfrac{N}{2} = \left\lfloor \dfrac{N}{2} \right\rfloor, & \text{if } N = even \text{ and } S = weak \quad (ii) \\ N - \dfrac{N+2}{2} = \dfrac{N-2}{2} = \left\lfloor \dfrac{N}{2} \right\rfloor - 1, & \text{if } N = even \text{ and } S = strong \quad (iii) \end{cases}$$

The value of $\left\lfloor \frac{N}{2} \right\rfloor$ is easily produced by shifting right by 1 bit position the 20-bit parameter N. So, on every cycle, the decrementor's value is compared to the number $\left\lfloor \frac{N}{2} \right\rfloor$; if the comparison yields equal for N =odd *or* $S=$ weak, then an *rclk* pulse is produced. In the case of N=even *and* $S=$ strong, an *rclk* pulse is produced in the following clock cycle, provided that one more sample of the same value is seen. If an incrementor were used instead of the decrementor *cnt*, then we would have to calculate the value of $K=\frac{N+1}{2}$, which requires an addition. The circuit implementing FSM operating on 1 sample per cycle is shown in figure 3.14.

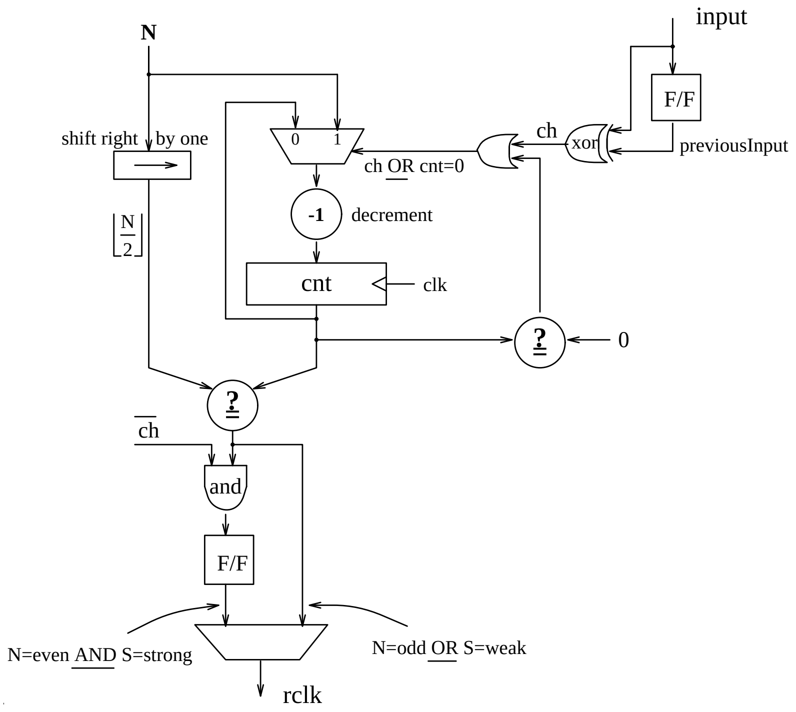


Figure 3.14: Circuit of the simple Receiver's Bit FSM, operating on 1 sample per clock cycle

A note should be made on an alternative solution to the counting sequence. In the current implementation the decrementor is initialized to $N-1$, counts down to 0, while when K samples have been seen (cnt=$N-K$) it outputs an *rclk* pulse. An alternative solution would be to load the decrementor to $K-1$, and count down to 0 for the first bit of a sequence, while for the subsequent bits, it would load the decrementor with $N-1$ and count down to 0. This alternative solution produces the *rclk* pulse when the decrementor reaches 0. It has the advantage that it compares *cnt* to 0, which is implemented by a large NOR gate, instead of using a comparator as in the current implementation. However, it needs an additional 20-bit multiplexor to select between K and N in order to load the *cnt* decrementor.

3.5.2 The Receiver Bit FSM operating on 2 samples per clock period.

This FSM, i.e. the one implemented in the chip, operates on two samples per clock period. The main idea in the two-bit FSM implementation is to combine pairs of columns of the table 3.2 so that the values of the decrementor as well as the decisions made are the same as those of the simple FSM previously described. The patterns of the table 3.2 which produce an *rclk* pulse are summarized in figure 3.15. Note that in case (b), the rclk pulse may equivalently be produced in the cycle $i+1$ instead of cycle $i+2$, since the current sample will cause *cnt* to decrement by one.

cycle	*i*	*i+1*
input	S	
cnt		$\lfloor \frac{N}{2} \rfloor$
decision		rclk

(a) N=odd OR S=weak

cycle	*i*	*i+1*	*i+2*
input	S	S	
cnt		$\lfloor \frac{N}{2} \rfloor$	$\lfloor \frac{N}{2} \rfloor$-1
decision			rclk

(b) N=even AND S=strong

Figure 3.15: Input patterns which result to rclk.

Since, the FSM operates on two samples, the counter should count for one or two samples, depending on how many identical samples are received. The problem that arises here is that it may skip the critical value of cnt_{rclk}, where the *rclk* is produced, e.g. for N=odd, it may count directly from $N-K+1$ to $N-K-1$. This problem is solved by checking the *cnt* for ≤ against $N-K$, instead of =. However, the operation of ≤ is equivalent to an addition. In order to avoid this full comparison which has a propagation chain of 20 bits and would further delay the time critical FSM operation, we used the following optimization: we check for equality of the decrementor to $N-K$ and any of the values $N-K+1$ or $N-K-1$. If equality to $N-K$ is found, the cases (i) and (ii) in the above formula for cnt_{rclk} are satisfied, and if equality to $N-K-1$ is found, all three cases are satisfied. Also, if equality to $N-K+1$ is found and there is no transition in the input, in the next cycle we know that we will reach a count of $N-K-1$. For the above comparisons, only one comparator and some small logic is required. One of the numbers $N-K-1$ and $N-K+1$ can be found without substraction: it differs from $N-K$ at the LS bit position. Thus, if at a given cycle, we detect the equality of *cnt* to $N-K+1$, then in the next cycle we will have a count of $N-K$ or $N-K-1$ depending on the amount of decrement (1 or 2).

Another situation that must be considered is when we have not yet decided on the current bit and there is a transition at the second sample. The first sample may be critical for deciding for the current bit, and it would be wrong to neglect it. For example, let us assume that a bit of value

S is being received, and that the samples acquired during this cycle are $[S,\bar{S}]$, where S is a sample of the same value as the current bit, and $\bar{S}$ is a sample of the different value. Let us further assume that no decision has been made, or can be made in this cycle for the current bit. Then, the question is whether in this cycle we should keep on with the current bit by counting the S sample, or initialize the reception of a new bit of value $\bar{S}$. In these cases, the FSM counts for the first sample by decrementing by 1, and carries the second sample to be counted in the next period, by setting the special flag, *Carry*. The flag indicates that "one sample from the previous cycle remained to be counted with the samples of the current cycle".

Table 3.3 presents the cases where an *rclk* pulse is produced, based on the FSM current state and the received samples. In this table, an "X" stands for "don't care".

Table 3.3: Receiver's Two Sample per cycle FSM decision on *rclk*

Current State			Samples seen now	Decision
cnt value	Value of S	Carry		
$=N-K+1$	Weak	X	S,X $\bar{S},X$	one S bit seen -
	Strong	X	S,X $\bar{S},X$	one S bit seen *if* N odd -
$=N-K$	Weak	X	X,X	one S bit seen
	Strong	0	S,X $\bar{S},X$	one S bit seen one S bit seen IF N=odd
		1	X,X	one S bit seen IF N=odd
$=N-K-1$	X	X	X,X	one S bit seen

In some cases, the conversion from the single sample FSM to the two-sample FSM presents some conflicts. These cases are shown in figure 3.16, and they both occur for the minimum bit length N=2.

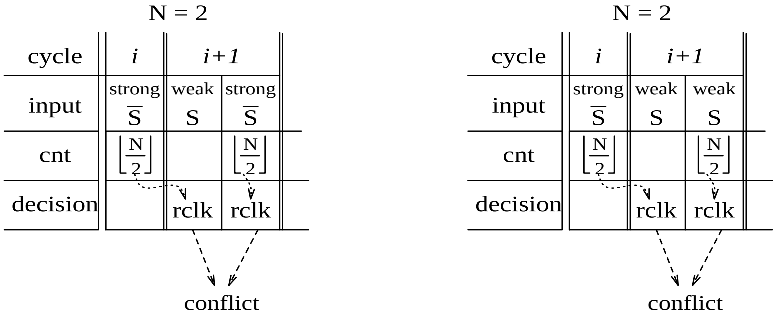


Figure 3.16: Conflicts in conversion from 1 sample/cycle to two sample/cycle FSM.

The conflict is presented because the decision that an $\bar{S}$ strong bit has been received is made on cycle $i+1$, and in the same clock cycle the single sample/cycle FSM decides that a weak bit S was also received, since it has seen $N-1=\left\lfloor \frac{N}{2} \right\rfloor=1$ samples of S. The result is that two *rclk* pulses should be output for this clock period. In our implementation, at clock cycle $i+1$, the *rclk* pulse corresponding to the first strong bit $\overline{S}$ is output, and the FSM starts counting the samples of the *S* bit. In the first case of figure 3.16, the decrementor will count for the *S* sample, and the *Carry* signal will be asserted signifying that an $\overline{S}$ sample remain to be counted with the samples of the next cycle; in the second case, the decrementor will count for the two *S* samples. In both cases, an *rclk* pulse corresponding to the *S* bit will be output on the next cycle $i+2$. If "short pulses" consisting of 1 weak sample and 2 strong samples continue being received, then the *Carry* bit will "overflow" and some incoming bit will be lost, since we cannot receive at a rate of >1 bits/clock period.

Figure 3.17 presents the circuit implementing the FSM operating on 2 samples per clock cycle. The *next state logic* block decides on the next FSM state. Its operation is given in the following table 3.4. This logic decides when to initialize a new bit reception when a change of the input is seen, and it also sets the *Carry* bit. The value of the current bit is toggled when an change of the input is observed. The decision logic block, described previously in table 3.3, decides if a bit has been received and outputs an *rclk* pulse together with the corresponding *bit value*. It also sets a latch, to prevent next false *rclk* pulses for the same bit. The decrementor, impelemented as a carry look ahead decrementor in stages of 4 bits, decrements by 1, 2, or 3 depending on the signals decr1 and decr2; if both signals are asserted, it decrements by 3.

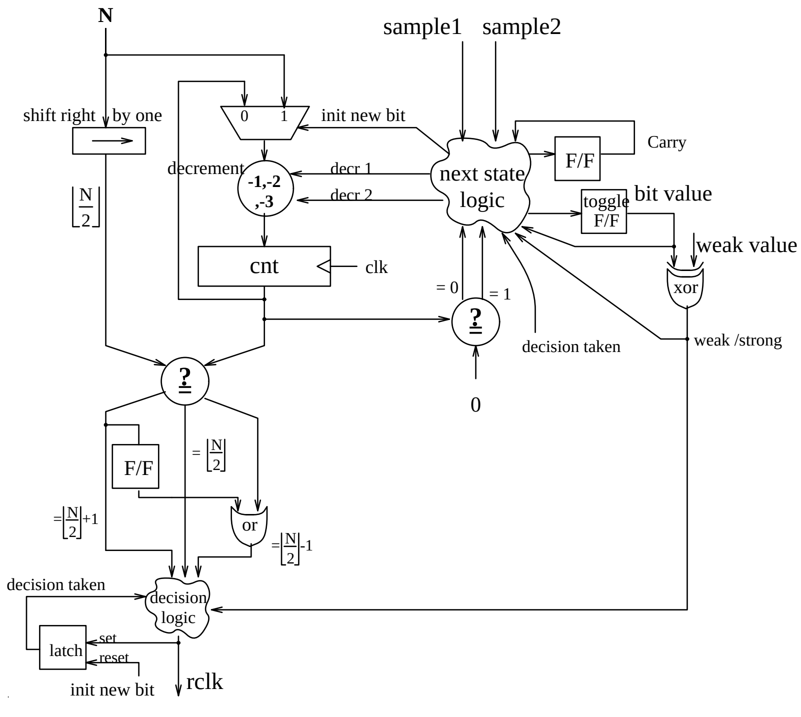


Figure 3.17: Circuit of the Receiver 2 sample/cycle Bit FSM

The next table 3.4 shows the next state of the FSM, implemented by the *nextstate* block in figure 3.17, based on the current state, the received samples, and the value of the Carry bit. Note that in some cases, the decrementor decrements by 3; this is when a *Carry* bit is of the same value as the two currently received samples, e.g. the entry in the table 3.4 where $n \le k-2$, S=weak, *Carry*=1, and the samples seen now are $\overline{S}$, $\overline{S}$.

Table 3.4: Receiver's Two Sample per cycle FSM transitions

Current State			Samples seen now	Next State		
Value of cnt	value of S	Carry		Sample value	Value of cnt	Carry
$>N-K+1$	Weak	0	$S,\underline{S}$	S	$cnt-2$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$cnt-1$	1
			$\underline{S},S$	S	$N-2$	0
			S,S	S	$cnt-2$	0
		1	$S,\underline{S}$	S	$cnt-3$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$cnt-2$	1
			$\underline{S},S$	$\underline{S}$	$N-3$	0
			S,S	S	$N-2$	1
	Strong	0	$S,\underline{S}$	$\underline{S}$	$cnt-2$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	$\underline{S}$	$N-2$	0
			S,S	$\underline{S}$	$N-1$	1
		1	$S,\underline{S}$	$\underline{S}$	$N-1$	1
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	$\underline{S}$	$N-3$	0
			S,S	S	$N-2$	1
$=N-K+1$	Weak	0	$S,\underline{S}$	$\underline{S}$	$cnt-2$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	S	$N-2$	0
			S,S	S	$cnt-2$	0
		1	same as for $n \le k-2$			
	Strong	0	same as for $n \le k-2$			
		1	$S,\underline{S}$	$\underline{S}$	$N-1$	1
			$\underline{S},\underline{S}$	$\underline{S}$	$N-2$	1
			$\underline{S},S$	$\underline{S}$	$N-3$	0
			S,S	S	$N-2$	1
$=N-K$	Weak	0	same as for $n = k-1$			
		1	$S,\underline{S}$	$\underline{S}$	$cnt-3$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	$\underline{S}$	$N-3$	0
			S,S	S	$N-2$	1
	Strong	X	same as for $n \le k-2$			
$=N-K-1$	same as for $n=k$					

Table 3.4: Receiver's Two Sample per cycle FSM transitions (continued)

Current State: Value of *cnt*	Current State: Value of *S*	Current State: Carry	Samples seen now	Next State: Sample value	Next State: Value of *cnt*	Next State: Carry
=1	Weak	0	$S,\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	S	$N-2$	0
			S,S	S	$N-1$	0
		1	$S,\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	$\underline{S}$	$N-3$	0
			S,S	S	$N-2$	1
	Strong	0	$S,\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	$\underline{S}$	$N-2$	0
			S,S	S	$N-1$	1
		1	same as for $n \le k-2$			
=0	Weak	0	$S,\underline{S}$	S	$N-2$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	1
			$\underline{S},S$	S	$N-2$	0
			S,S	S	$N-1$	0
		1	$S,\underline{S}$	S	$N-3$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	1
			$\underline{S},S$	$\underline{S}$	$N-3$	0
			S,S	S	$N-2$	1
	Strong	0	$S,\underline{S}$	$\underline{S}$	$N-2$	0
			$\underline{S},\underline{S}$	$\underline{S}$	$N-1$	0
			$\underline{S},S$	$\underline{S}$	$N-2$	0
			S,S	S	$N-1$	1
		1	same as for $n \le k-2$			

Chapter 4

Design Methodology and Post-Fabrication Testing

In this chapter we discuss the general design methodology that was followed throughout the design of the chip. The design flow is presented in section 1, while section 2 gives the main characteristics of the macrocell and describes the design of the power distribution network. In section 3, we give details on the simulations, fault simulations, and patterns for test of the chip, while in section 4, we describe the post fabrication testing setup and the performed tests. Finally, in section 5, we evaluate the design environment and present some general thoughts on CAD tools for VLSI.

4.1 Design Flow

Our chip was designed using the Cadence Design Environment, and the European Silicon Structures (ES2) SOLO2030 configuration for its 1.5 μm process. First, the schematic entry together with functional simulations were performed. These two tasks are interrelated, since the simulation tests the correct operation of the circuit and serves as a guide for changes in the schematic. The design was built using symbol parts from the ES2 ecpd15 library, in an hierarchical form. Each subpart was simulated separately for correctness, before being used in the design. Also, a library with macrocells was developed. When the whole design was complete, and its operation was verified through extensive simulations, fault simulation was performed to generate vectors for post-fabrication testing (see also § 4.3 on verification).

The next step was the placement and routing. We ‘‘flattened’’ the hierarchical schematic, and translated all symbols from the library into their *abstract* representations. The *abstract* representation is an intermediate model between the symbol of a cell used in a schematic, and the layout of this cell: it contains the outline and the external connections of the layout cell, that is, all information needed for the placement routines and routers. Besides the abstract representations of the library cells, the flattened design included also the variable-input-threshold cell which was independently laid out by hand. The next step was to assign high priorities to the critical signals, that is clocks, reset signals, and some signals in the receiver's FSM which go through a long path. Also, the widths of the important nets were specified as a multiple of the default width of the metal in which the signal is routed. We assigned a width of 40 to the core ground and power signals, and 2.3 to the external clock signals, and 2.2 to the generated clock signals. The clock signals were assigned a width twice the default one, because the default width is only suitable for short distance interconnects that carry very small currents. Since the clock wires are distributed and used all over the chip, the load capacitance of the driven transistors is much greater than the wire capacitance. A wider wire is needed in order to carry the current of the clock drivers.

For the placement phase, we defined regions and groups of signals that go to each region. The placement tool of the Cadence system does ''comprehensive placement'', that is, it finds locations of components as well as the assignments of nets to pins and logical gates to physical gates, [PrLo88]. For each region, certain parameters are defined, such as number of rows, height of each row, and alignment of cells. One or two preliminary runs of the placement tool, give a ''feel'' for the congestion areas and appropriate values for the placement parameters. The routing was done in two steps. First the global router was used to define routing regions, the channels, and allocate specific nets to each channel. The global router implemented in the Cadence environment is a hierarchical router, that works on the tree of horizontal and vertical channels. The global routing is done in a combined manual and automatic way. The supply nets, clocks, reset, and critical signals to the internal circuits of the chip were routed manually, while the rest of the nets were routed automatically. The next step was the detailed routing, which created the physical geometries in the channels according to the assignments specified by the global router. The detail router creates the connections and vias required. It routes each channel independently from the others, and it may compact or expand the channel to route all signals specified by the global routing phase. Once the physical layout of the chip has been completed, the actual sizes of the routed signals were known, and the paracitic capacitances could be extracted. We used the values of the capacitances to run a full ''loaded'' simulation with the same test stimuli as before.

The foundry interface is a set of utilities to prepare the design for manufacturing. It includes generation of vectors for operational simulation, tester interface, that is translation of the vectors to the tester format, and packaging utilities. The final outcome of the process of fault simulation are the test vectors and the results of the simulation, i.e. the values of the output signals for each test period. Besides the operational simulations ran at different process speeds (min, typ, max), two faults simulations were run, one with slow silicon and the other with fast, and their results were compared to verify that the test vectors discover all possible faults in all ranges of speeds. The test vectors and the corresponding expected outputs were translated to the tester format, in order to be applied by the tester onto the fabricated device to verify if it is a fault-free or faulty die. Finally, using the Packaging utilities of the SOLO2030, we chose a suitable package for the design and checked the pad ring power and ground connections.

4.2 Macrocell Characteristics

The macrocell designed consists of about 1500 gates, of which 650 are for the transmitter and 850 for the Receiver, or equivalently 2600 transistors for the transmitter, and 3400 transistors for the receiver. The chip that contains the transmitter and receiver blocks, is a square of side 4.15 mm, and is shown in figure 4.1. The core area dimensions are about 2.9 mm on each side making a total of 8.4 mm^2, while the area occupied by the pads is 2 mm^2. Inside the core, the receiver occupies is 2.7 mm^2, while the transmitter takes 1.8 mm^2. The rest of the core area is comprised by the select logic that arbitrates the 8-bit I/O bus †, 0.2 mm^2, by the tristate drivers of the output signals on the 8-bit I/O bus, 1.3 mm^2, and by interconnect space. The receiver cells are placed in 10 rows, and the transmitter cells are ordered in 7 rows. The drivers block is made up of 21 rows of cells. The percentage of the chip area that is dedicated to interconnect is about 45%.

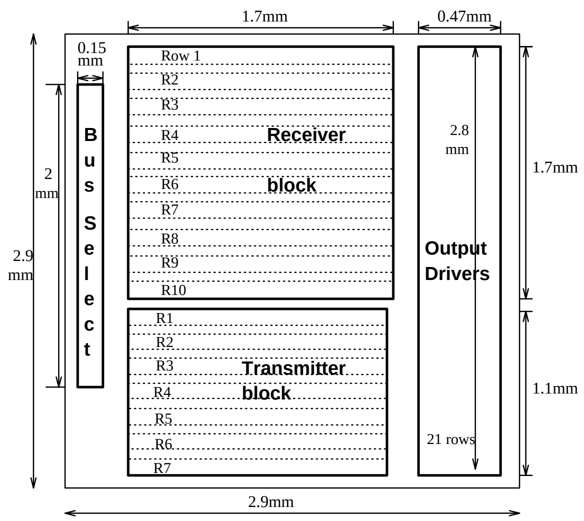


Figure 4.1: Chip floorplan (without I/O pads).

The pad ordering and the power and ground supply network are shown in figure 4.2. The chip is packaged in a 28-pin DIL. Four of the pads are dedicated for ground and power supplies for the core and the output pads separately. The remaining pads are:

This 8-bit bus is used to load parameter values and to provide a path to the internal points of the chip for the purposes of testing.

Table 4.1: Description of the chip pads				
Number of pads	I/O	Name	Polarity	Description
5	Input	Bsel[0:4]	high	bus select signals
8	I/O	IObus[0:7]	low	I/O pads for the byte-wide bus
1	Input	Bio	high for input	pad that determines the direction of the bus pads
1	Input	Xclk	high	Transmitter's external clock source
1	Input	Xreset	low	Transmitter's reset
1	Output	Serout	low	Serial Output of the Transmitter
1	Output	Xsh_empty	high	Transmitter's shifter empty
1	Output	Xfull	high	Transmitter's hold buffer full
1	Input	Rclk	high	Receiver's external clock source
1	Input	Rreset	low	Receiver's reset
1	Input	Serin	high	Serial input
1	Output	R_rdy	low	there is one received character
1	Input	char_read	low	received character read by the external circuits

The pads that correspond to the receiver are placed on the left side of the chip, so they can be connected directly to the receiver block. The transmitter's pads are placed on the bottom-left side of the chip, close to the transmitter block. The select pads are placed on the left side of the chip, because they are used in the select block that generates control for both the receiver and transmitter. Finally, the I/O pads of the 8-bit bus, which are larger than the other pads, are placed on the right and bottom sides.

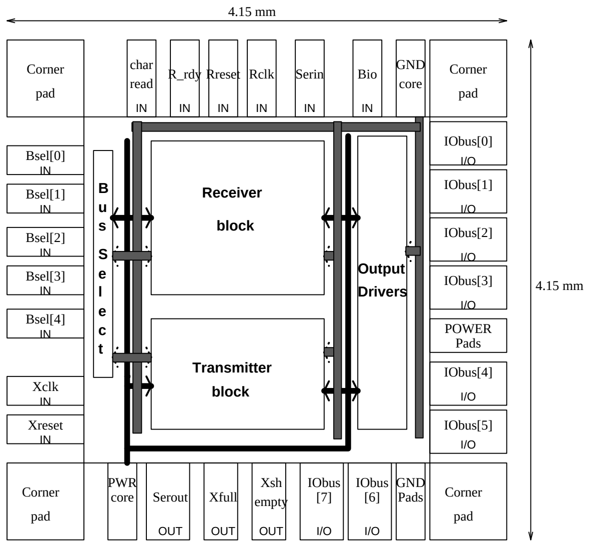


Figure 4.2: Pad arrangement and supply network of the UART chip.

Separate power and ground cells have been used to provide power supplies to the input/output pads and the core of the chip. The benefits of this configuration is that output buffer switching noise does not affect the core and input buffers. The core supply nets are placed in ‘‘comb’’ style, where the main lines are drawn in the vertical direction, and the secondary horizontal lines supply the rows of components. The supply lines inside a row run through the cells since the supply pins of adjacent cells overlap. For the large central blocks, we have used supplies from both left and right sides, so that the voltage drop of the supply track along a component row is minimized.

4.3 Analysis and Verification

Two of the steps of digital design are circuit analysis and circuit verification. Analysis is the process of modeling the behavior of the circuit and testing whether the circuit behaves according to the requirements. In this process we check the logic (logic simulation), as well as the critical paths and timings (timing simulation). Verification is the process of generating the test patterns that will be applied after fabrication to verify that the physical circuit behaves the same as its simulated representation. This is the step where we check the correctness of the manufactured circuit.

For our chip the logic simulation was done using mainly the Silos gate level simulator [Silos89]. Since Silos is a very low-level simulator, taking input in the form of test vectors only, we have used the Cadence STL interpreter as a front end. STL is a high level description language for the generation of test vectors, and provides variables, iteration, conditional statements and procedures. The about 10.000 test vectors tested in detail the behavior of the receiver and transmitter for many combinations of the configurable parameters, and for expected input patterns. More specificly, we simulated the transmitter’s behavior for bit length values of 2, 3, 4, 15, 32, and 1356, for all nominal values of character length and parity, for pipelined transmission of successive characters, and for asynchronous loading of characters. Similarly, the receiver was simulated for bit lengths of 2, 3, 4, 5, 21, and 1022, all valid values of character length and parity, and for distortion of the input signal of 33% in case of bit length equal to 2, up to 50% for bit length equal to 1022. In the receiver’s simulation we did not include in the path the Variable Threshold Unit, since we had already tested it through spice simulation, and modeling it in Silos would very complicated. Additional simulations were carried out using the Verilog gate level simulator [Gate89] for the 1.0 μm standard cell library of ATMEL [Atmel90]. In these simulations, we used two representations of the UART cell: the netlist of the schematic using the models of the basic cells supplied by ATMEL extracted for Verilog, and a hand-built behavioural model of the receiver and of the transmitter. We applied the same test stimulus concurrently, to both the behavioural and the functional model, and verified that in all cases we get the same results. Apart from evaluating our design with another manufacturing process, the flexibility and power of the Verilog simulator helped us in debugging the early design.

Timing simulations were ran after the placement and routing of the design. In the early stages of the design, the timings of the logic simulations were not exact, since they were based on the estimated delays due to interconnect loads, that are built in the Silos models of the components. The final lengths and widths of the wires are made known after the placement and routing. In this stage, the exact capacitive and resistive loads of the wires can be computed, and the true delays are known. The critical timings were identified by inspecting the graphical waveform output, since the Silos simulator does not have any advanced timing capabilities. The most time critical part of the circuit was the receiver's FSM. By successive placement+routing runs followed by loaded timing simulations, we found the better placement in terms of compactness and performance.

The process of verification involved two tasks, and the addition of specific hardware that provides access to internal paths, and the fault simulation of the circuit. *Fault simulation* of a digital circuit is the modeling of the network's behavior in the presence of faults which can be caused by physical defects or environmental influences. To make the simulation concrete, some specific assumptions are made about the faults present; this step is known as ‘‘fault insertion’’. We performed fault simulation using the Fsilos simulator, an extension to the Silos simulator, together with the fault models for the parts of the ES2 library. Fsilos follows the gate-level ‘‘single stuck-at’’ model, which is the fault-model most commonly used. This model assumes that each net in the circuit is stuck-at either the logic 1 or the logic 0, and such a fault is the only one in the circuit. For each fault, a simulation is run. The fault is said to be *detectable*, if the output of the simulation with the specific fault differs from that of the fault-free simulation. The model limits the number of faults to one at a time, for reasons of computability; for a network containing N nodes, where each node can be in one of the 3 states, ‘‘stuck-at-1’’, ‘‘stuck-at-0’’ and fault-free, 3^N cases should have to be considered if this simplification were not made!

The aim of the test-generation process is to derive a set of vectors that covers as many faults as possible. We measure the effectiveness of the test vectors by calculating the *fault coverage* which is defined as the ratio of the number of faults detectable by the patterns to the total number of faults assumed present in the circuit. Thus, we are trying to reach 100% fault coverage, although in many cases a lower percentage may be acceptable, due to practical difficulties in testing. In the fault simulations that we run, we used as a starting point the logic simulation patterns, and modified them appropriately in order for them to reach a high fault coverage. This involved extending the test vectors to include more combinations of the input data. By running *activity analysis*, which determines the number (and percentage) of the nets toggled by the simulation vectors, we had a first estimate of the effectiveness of our test stimuli, since the full fault simulation is a time consuming process. We obtained a fault coverage of 93% with our final test vectors. This percentage would increase if we triggered all states of the 20-bit counters. Therefore with the existing test length limitation of a maximum of 64000 test vectors, and with the need to have divisions of the clock cycle to simulate asynchronous signals, this was not feasible. Also, a higher fault coverage does not always result in fault diagnosis. By its nature, the existing fault model does not help fault diagnosis, since different faults can cause the same observable errors, and the most common failures are the gate-to-drain and gate-to-source short circuits, [Holl88].

The testability of digital circuits can be increased by adding special hardware. Many approaches have been proposed in literature, the main of them being: latch-scanning arrangements, e.g. scan-path, switching of I/O ports, and internal, pattern-generation and response compaction, e.g. built-in tests, [Tsui87]. In our design, we have used the simpler and clearer of these methods: switching of I/O ports. We used the already existing 8-bit I/O bus through which input parameters are loaded, and provided paths from internal points to this bus through tristate drivers. The pads with the select signals for the bus arbitration were increased to 5, and a 5-to-32 coder was used to select which path drives the bus. This approach was preferred to the scan-path circuitry, because it is clearer, and makes internal points immediately observable. Also in our case where the design was not pad limited, we could easily add 2 more pins to the initially needed 3 for the select logic.

4.4 Post Fabrication Testing

In this section we will describe the post fabrication tests that were performed until today (end of July 1992) and those that will be performed in the next weeks. Some preliminary results show that the chip operates successfully both at the data rate of 115.2 Kb/s when interfaced to a commercial UART chip, as well as at 20 Mb/s in loopback mode through a 100m coaxial cable. The testing is following two independent plans: i) testing on a 50 MHz, 64-pin VLSI-tester brand Tektronix model LV500, and ii) developing a prototype board where a microprocessor is used to interface to our chip.

The prototype board includes one UART chip, the intel-8031 microcontroller, and some glue logic. The 8031 is responsible for the setup of the UART by writing to the chip registers the parameter values, for generating the signals to load a character for transmission in the UART transmitter, and for reading the received characters from the UART receiver. An external clock source is fed to the UART, and also serves as the microcontroller's clock divided by 2. We have tested the UART chip at 10 and 20 MHz, but we have not gone above 20 MHz, because of the limitation that the 8031 clock has to be a submultiple of 10 MHz, and of the difficulty in finding high frequency crystals.

In one of the tests performed, the UART is used in loopback mode. More specificly, we are sending individual characters through the serial port of a PC to the 8031, which in turn sends them to the UART transmitter. The serial output is fed to the serial input, and when a full character has been received, the UART interrupts the microcontroller which reads the character and gives it back to the PC.

In the second test, the loopback cable is a 100m coaxial cable of type RG58. An LH63 amplifier is used for converting TTL signals to drive 50 Ohms, while the raw serial input is directly fed to the receiver.

Another test was to connect the serial output of our chip directly to the serial port of the PC, and the inverse test where our UART receiver takes input from the PC's serial port. We verified that it worked successfully at the PC's maximum allowable data rate of 115.2 Kbaud. Finally, the fault testing on the VLSI tester will be done in the next weeks.

4.5 Evaluation of the Design Environment

In this section we will evaluate the design environment, and present some thoughts in the area of CAD systems for VLSI. All stages of the design were done under the Cadence OPUS design system. The Cadence environment is one of the latest and most complete design systems currently available for VLSI design. The philosophy of large design systems like Cadence is the integration of tools under a unified environment, and provision of a general framework that can be configured according to the wills of the user. The OPUS system contains a schematic editor, extraction tools, supports many simulators like Silos, Spice, Hspice, Verilog, and is open to be configured to support any other simulator. It also provides placement tools, routers, layout editor, symbolic layout, physical design verification tools, and others. Thus, it is obvious that it supplies with all the tools that a designer needs. However, the extended configurability is obtained at the expense of simplicity. It takes a lot of time to learn how to use the system. One has to go through dozens of manuals, and to cross check redundant pieces of information in them, in order to find out how a simple job is done. On the other hand there are tools like "Magic" (of the Berkeley VLSI Tools Suite), which may not be so sophisticated but are easy-to-use and powerful. As a conclusion, we would prefer a CAD system having all the tools integrated, but retaining its simplicity.

Some additional notes on the weak points of the system follow, focusing on the layout editor and the simulation process. The layout editor does not have automatic Design Rule Checking (DRC). The tool is optimized for automatic synthesis, and the orientation is to use symbolic layout, which resembles to stick diagrams without following exact design rules, and then compacting the design automatically. However, DRC is a very useful feature when the layout is done manually. Another weak point is the simulation process. In order to run a logic simulation, the design must be extracted to establish connectivity, the new design must be extracted to obtain its flattened netlist, the simulation is run and the results are shown graphically. This is a dull process, repeated many times when we do small changes in the schematic and we want to quickly get the results. A better system would include the capability, to automatically detect changes in the schematic, simulate and update its output, without making the user go through the above steps. This is realistic and can be done in real time, for small designs where the run time of the simulation is small.

In our design, we had to use the ES2 configuration that included the libraries, models for the Silos simulator, and many routines for netlist extraction, simulation, and foundry interface. This configuration was not well prepared, and it was full of errors, which we had to correct in order to continue our design. For example, the capacitances and resistances of metal as given in the technology files were wrong. Moreover, the "include" files for "loaded simulation" with the actual paracitic capacitances did not take them into account at all. If someone followed the ES2 user's manual without checking, absolutely wrong results would be obtained. Another drawback was that the library part models of the ecpd15 library that we used were given only for the Silos simulator, which is very primitive. Thus, we had to write the simulation stimuli in an intermediate language and then translate it to Silos. Configuring the system for the Verilog simulator, would have been a far better choice.

Chapter 5

Conclusions, Extensions

In the context of serial asynchronous communications, clock recovery by oversampling (e.g. 16x) is not applicable to high communication rates, while clock recovery using phase-locked loops (PLL) requires considerable silicon area and considerable design expertise. Our approach to data recovery using a minimal number of samples per bit (as few as 2) is simple and allows inexpensive serial communication at very high rates.

The major advantage of our cell over clock recovery using PLL is that it occupies a small area. Moreover, it can be implemented using standard cell components, and it can be easily ported to various technologies and processes with standard cells, without requiring high expertise and design effort. If we compare our cell with the commercially available UART cells, we find that our cell has better distortion tolerance, and thus performs better than other UARTs under noisy conditions. Assuming an equivalent setup, where a conventional UART uses a 16x clock and we use N=16 samples per bit, the conventional UART tolerates sampling error of 43.7% of the single pulse width, while our method tolerates 46.8% of the pulse width.

All the above features make our cell appropriate to be used as a building block in full-custom or semi-custom VLSI designs that need to interface between serial asynchronous communication links. One of the possible uses would be in interfacing with standard RS232 links, where the operation speed is limited by the other end device. In such cases, a standard 1488/89 driver/receiver pair for RS232 can be used, together with low cost cable. If our cell is used at both ends of the communication link, it can send and receive at its maximum rate. Then, a driver such as the LH63 amplifier could be used together with good quality, shielded, 50 Ohm coaxial cable that enables connections up to 100m.

Another use of our cell would be in building a fast and inexpensive network interface. Since the receiver and transmitter blocks are small in size, several of them can be integrated in one chip together with buffer memory and routing control, thus implementing the lower levels of a ring or star network topology with point to point links, or a combination of the above. As a conclusion, our cell is small and flexible, and can be used in a wide range of communication applications.

The work presented here can be extended in several directions, three of which are:

- a pulse shaping circuit for the clock
- balanced transmission of 1s and 0s
- application of the decoding algorithm to synchronous protocols

The clock signal that is fed to our chip may not have a duty cycle of 50%, that is its two half-periods may not be of equal length. A solution to this problem would be a pulse shaping circuit that shapes the clock signal. A preprocessing circuit analogous to the one presented in § 2.2 can be used to selectively expand or shrink one of the clock subperiods.

A future addition to the chip protocol would be to send a balanced number of 1s and 0s, in the long term. This would help with the ground reference problem, present when the two interconnected devices are physically situated apart one from another. In these cases, the reference ground voltage may be different in the two places, resulting to a current flowing between the two interconnected sites through the ground signal wire. This in its turn results to a voltage drop which may cause a different (wrong) interpretation of the signal applied by the transmitter driver at the receiver end. To solve this problem, a circuit with decoupling capacitors (or an equivalent with inductors) can be used for semi-differential mode of transmission. The capacitor lets only the ac-part of the signal pass through it, while the dc-part remains stable. A problem arises when many 0s or 1s are sent: the load which accumulates in the capacitor biases the dc-part of the signal, and limits the voltage swing. If the transmitter could send in the long term a balanced number if 1s and 0s this problem would be solved. This can be implemented by sending a special synchronization pattern of successive 1s and 0s at the idle periods.

Our decoding algorithm is of general nature and can also be applied to synchronous protocols, such as Ethernet (Manchester encoding), FDDI (4B/5B encoding), and X-25 (bit-stuffing) [Walr91], etc. In the Machester encoding scheme, the receiver synchronizes its clock using the transition inside the bit interval. The 4B/5B and the bit-stuffing codes, use more bits than those necessary to carry the information in order to ensure that there is at least one transition in groups of 3 bits or 5 bits respectively. The transition is necessary to keep the receiver's PLL locked. Here also, our decoding algorithm can be used for data recovery, instead of the PLL. Thus, our decoding algorithm can be an alternate solution to the synchronization by PLL, having the advantage of small and easy implementation.

References

[Atmel90] ‘‘ATL Series 1.0 µm CMOS Gate Array Design Manual’’, Atmel Corporation, 1990.

[Best85] R. Best: ‘‘Phase Locked Loops’’, 1985.

[ESS90] ‘‘SOLO 2000 Family Libraries’’, European Silicon Structures, June 1990.

[Gate89] ‘‘Verilog-XL Reference Manual’’, Gateway Co., 1989.

[Hama86] G. Hamachi: ‘‘Designing Finite State Machines with PEG’’, Berkeley CAD Tools User’s Manual, 1986.

[HaPu91] S. Hao and Y. Puqiang: ‘‘A high-speed Digital Phase-Locked Loop’’, IEEE Journal of Solid-State Circuits, vol. 39, no. 3, March 1991, pp. 365-368.

[Holl88] E. Hollis: ‘‘Design of VLSI Gate Array ICs’’, Prentice-Hall, 1988.

[JBHK87] D.K. Jeong, G. Borrielo, D.A. Hodges, R. Katz: ‘‘Design of PLL-Based Clock generation Circuits’’, IEEE Journal of Solid-State Circuits, vol. 22, no. 2, April 1987, pp. 255-261.

[JoHu88] M. Johnson and E. Hudson, ‘‘A variable Delay Line PLL for CPU-Coprocessor Syncronization’’, IEEE Journal of Solid-State Circuits, vol. 23, no. 5, Octomber 1988, pp. 1218-1223.

[McNam88] John McNamara: Technical Aspects of Data Communication, Digital Press, 1988.

[Mess90] D. Messerschmitt: ‘‘ Syncronization in Digital System Design’’, IEEE Journal on Selected Areas in Communications, vol. 8, no. 8, October 1990, pp. 1404-1419.

[Op83] Alan Oppenheim, Alan Willsky: ‘‘Signals ans Systems’’, Prentice-Hall, 1983.

[PrLo88] B. Press, M. Lorenzetti: ‘‘Physical Design Automation of VLSI Systems’’, Cummings Publishing Co., 1988

[Silos89] Cadence Design Systems: ‘‘Silos II’’, 1989.

[Spice] L. Nagel: ‘‘Spice2: A Computer Program to Simulate Semiconductor Circuits’’, ERL MEMO ERL-M520, University of California Berkeley, California 1975.

[Tsui87] F. Tsui: ‘‘LSI/VLSI Testability Design’’, McGraw-Hill Book Co., 1987.

[Uyem88] J. Uyemura: ‘‘Fundamentals of MOS Digital Integrated Circuits’’, Addison-Wesley Co., 1988.

[Walr91] J. Walrad: ‘‘Communication Networks’’, Aksen Associates Inc. Publishers, 1991.

[WeEs85] N. Weste and K. Eshragian: ‘‘Principles of CMOS VLSI Design’’, Addison-Wesley Publishing Co., 1985.

Appendix A

Appendix A presents the detailed schematic diagrams of the transmitter and receiver blocks.

Figure A.1: Overview of chip schematic

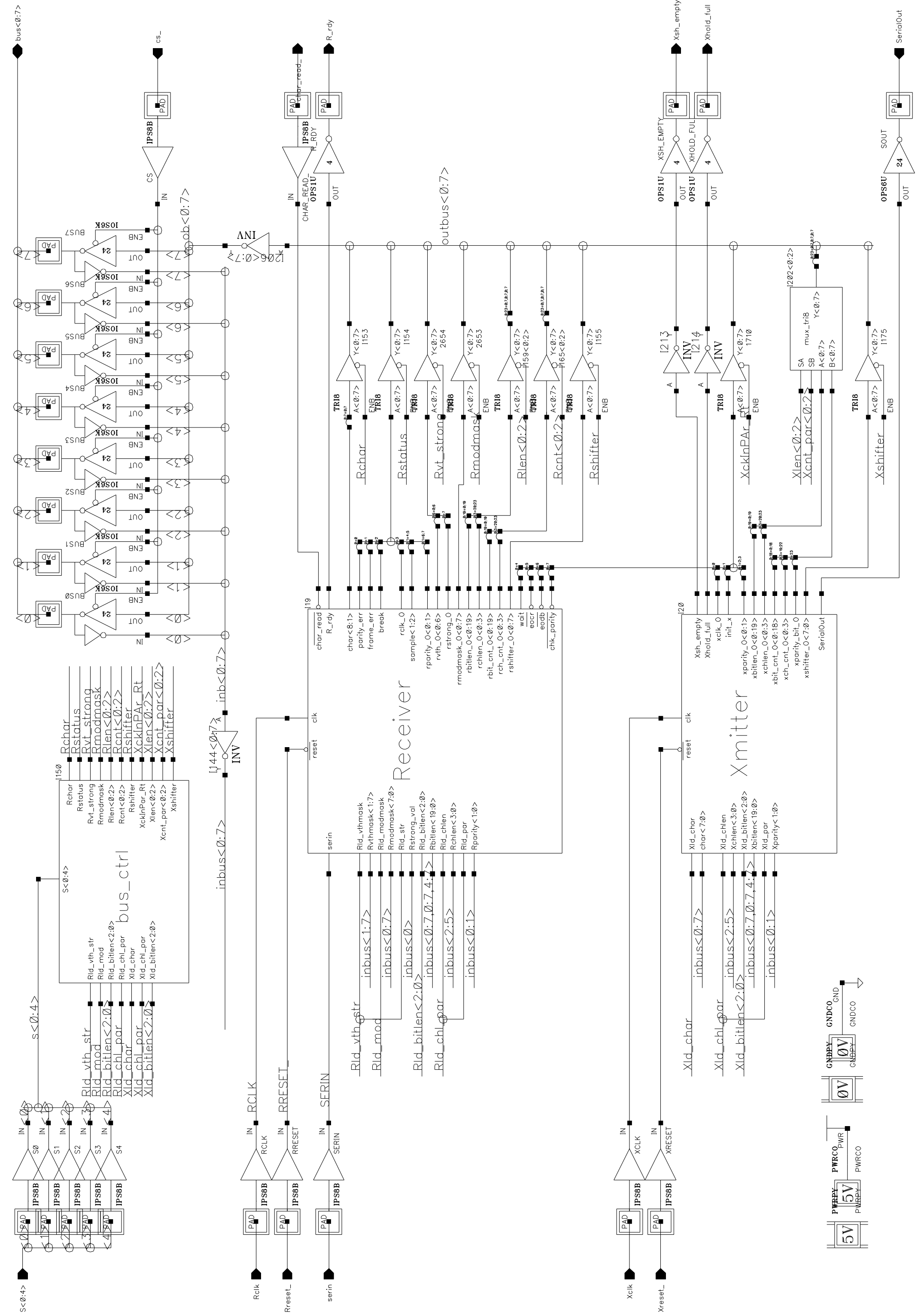


Figure A.2.1: Transmitter General Block Diagram

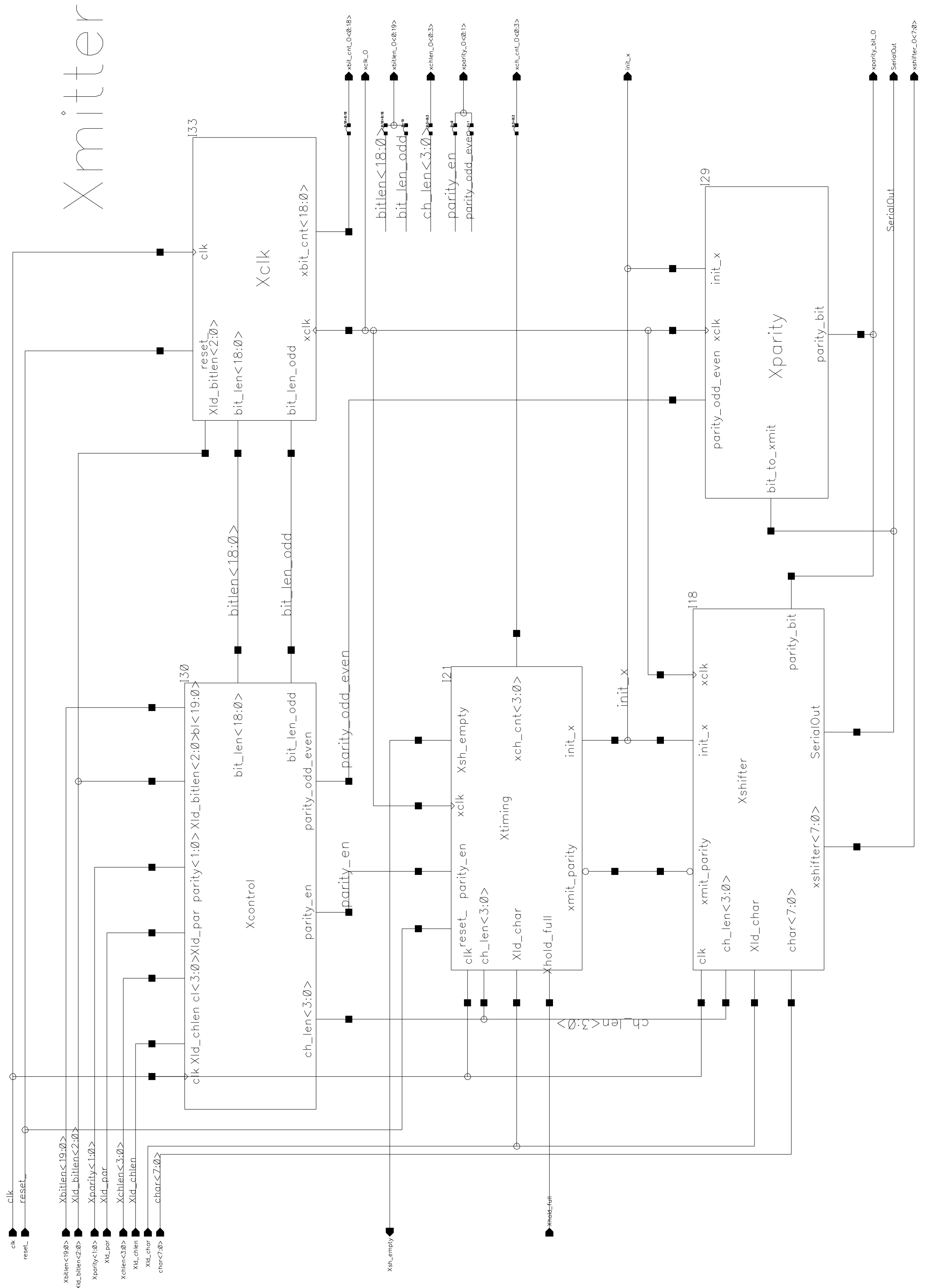


Figure A.2.2: Transmitter Circuit for Setting of Parameters

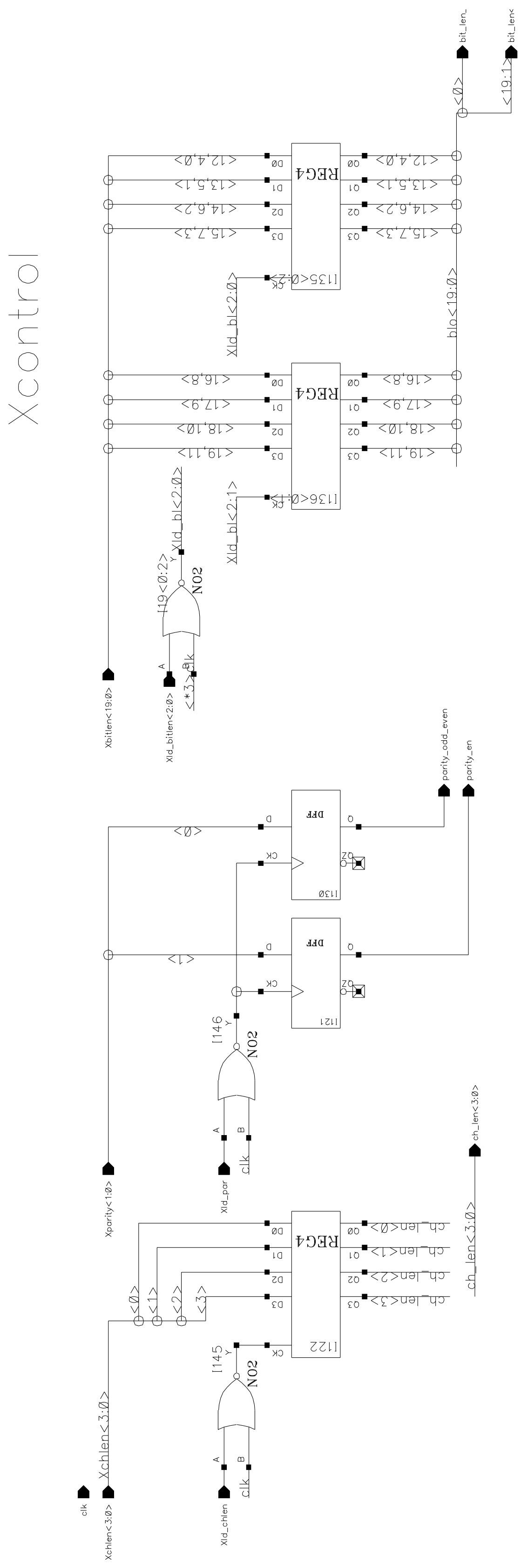


Figure A.2.3: Transmitter Clock Generation Circuit

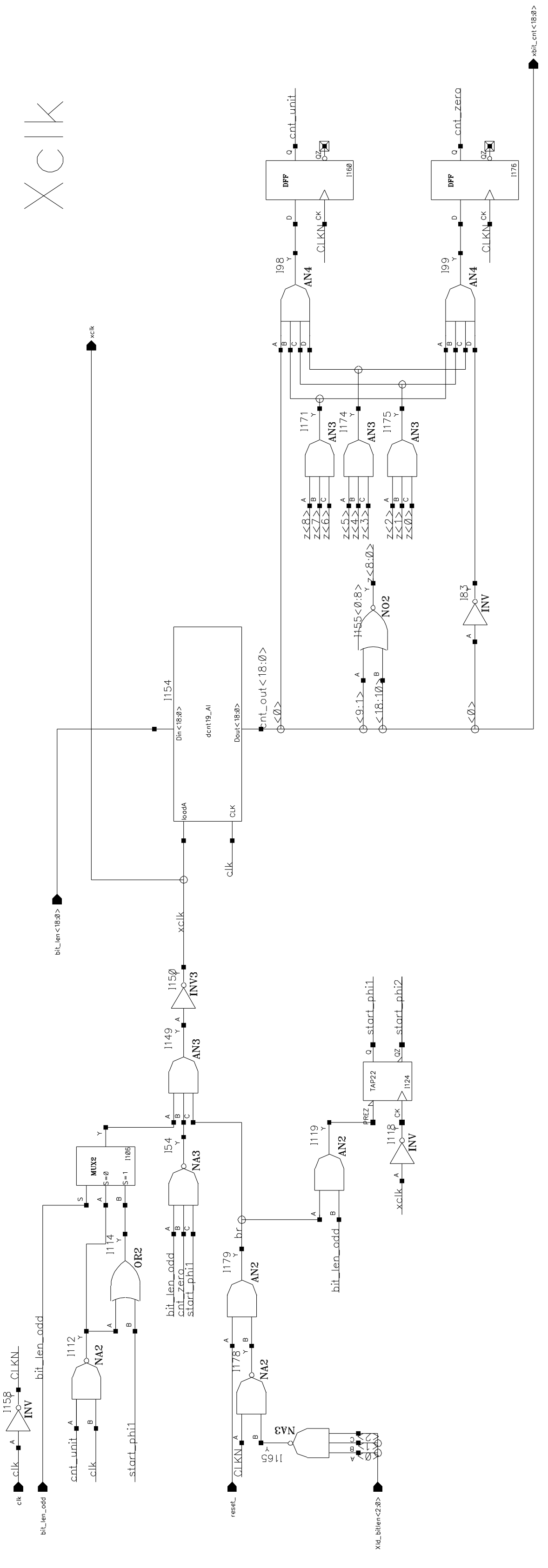


Figure A.2.4: Transmitter Timing Signals Circuit

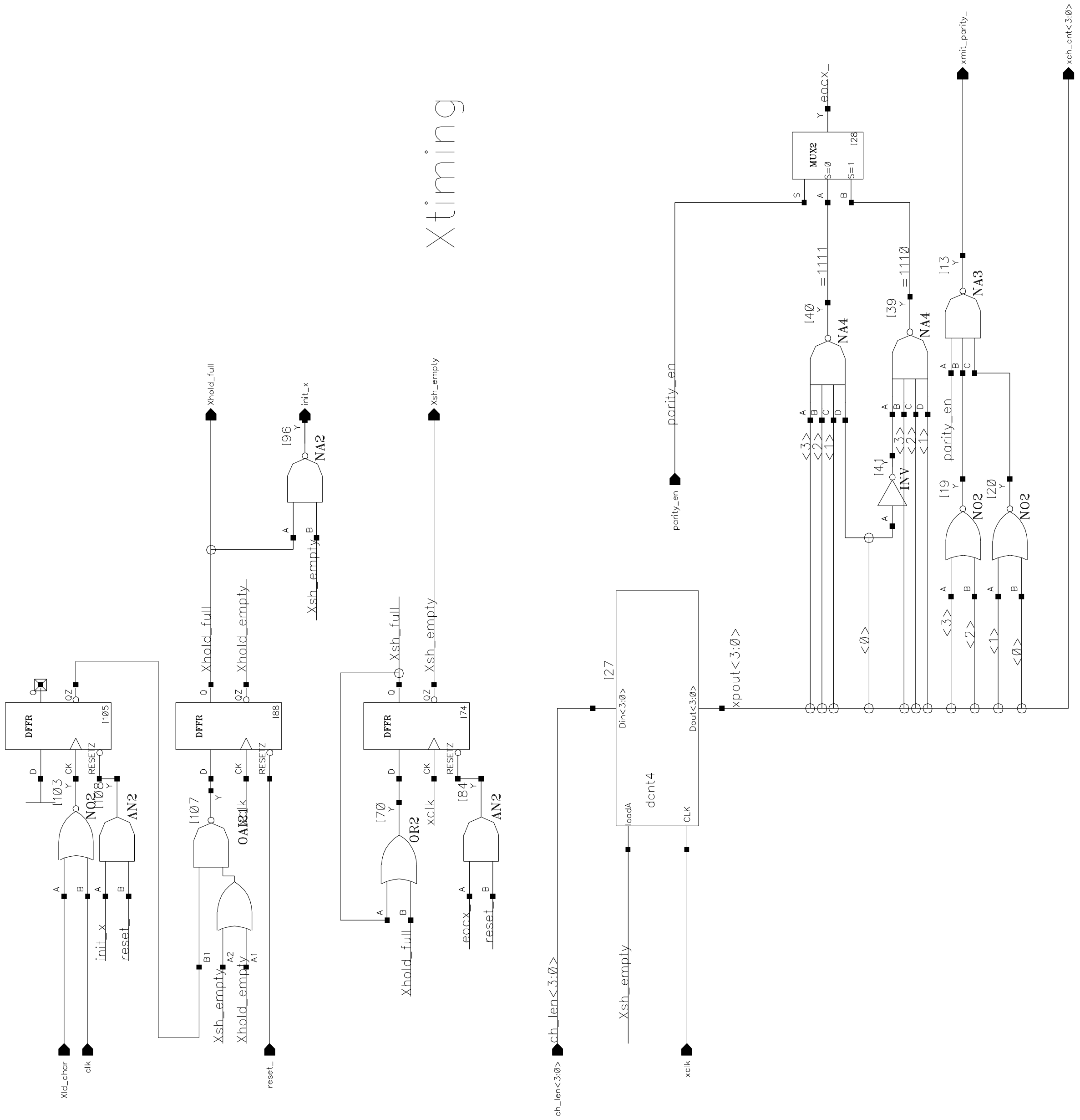


Figure A.2.5: Transmitter Shifter Register Circuit

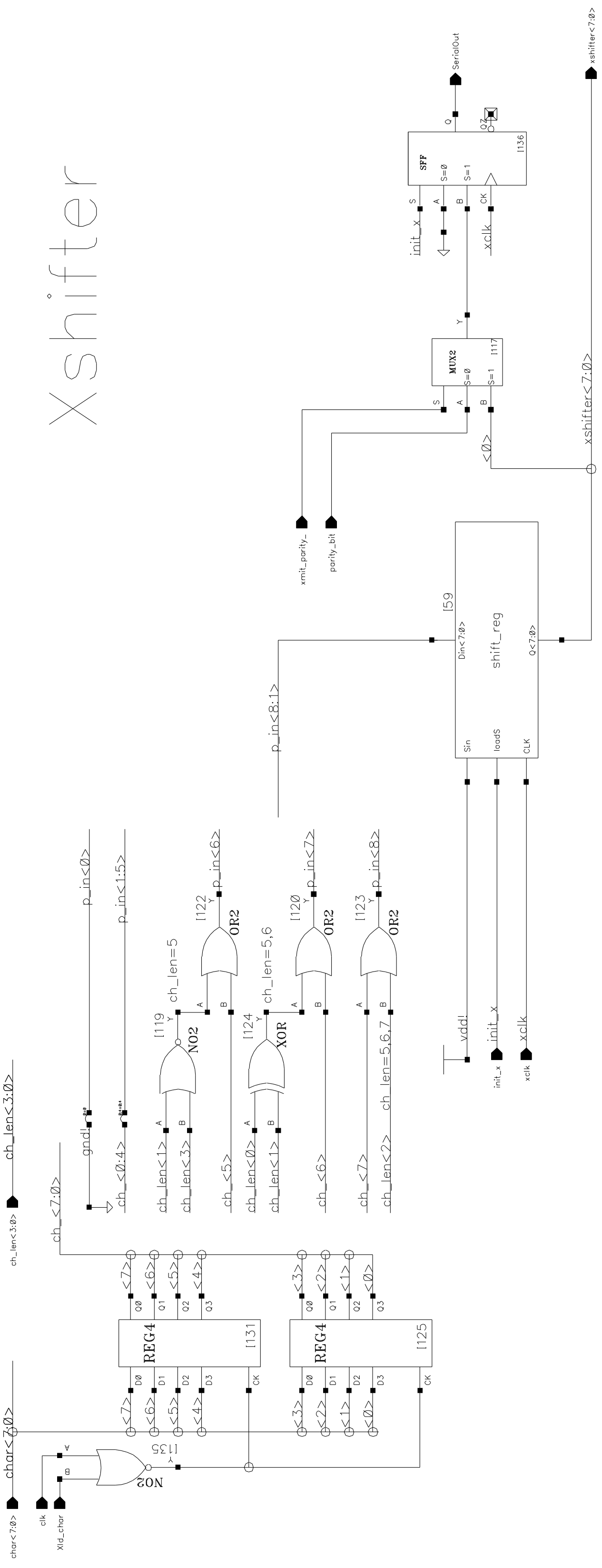


Figure A.2.6: Transmitter Parity Generation Circuit

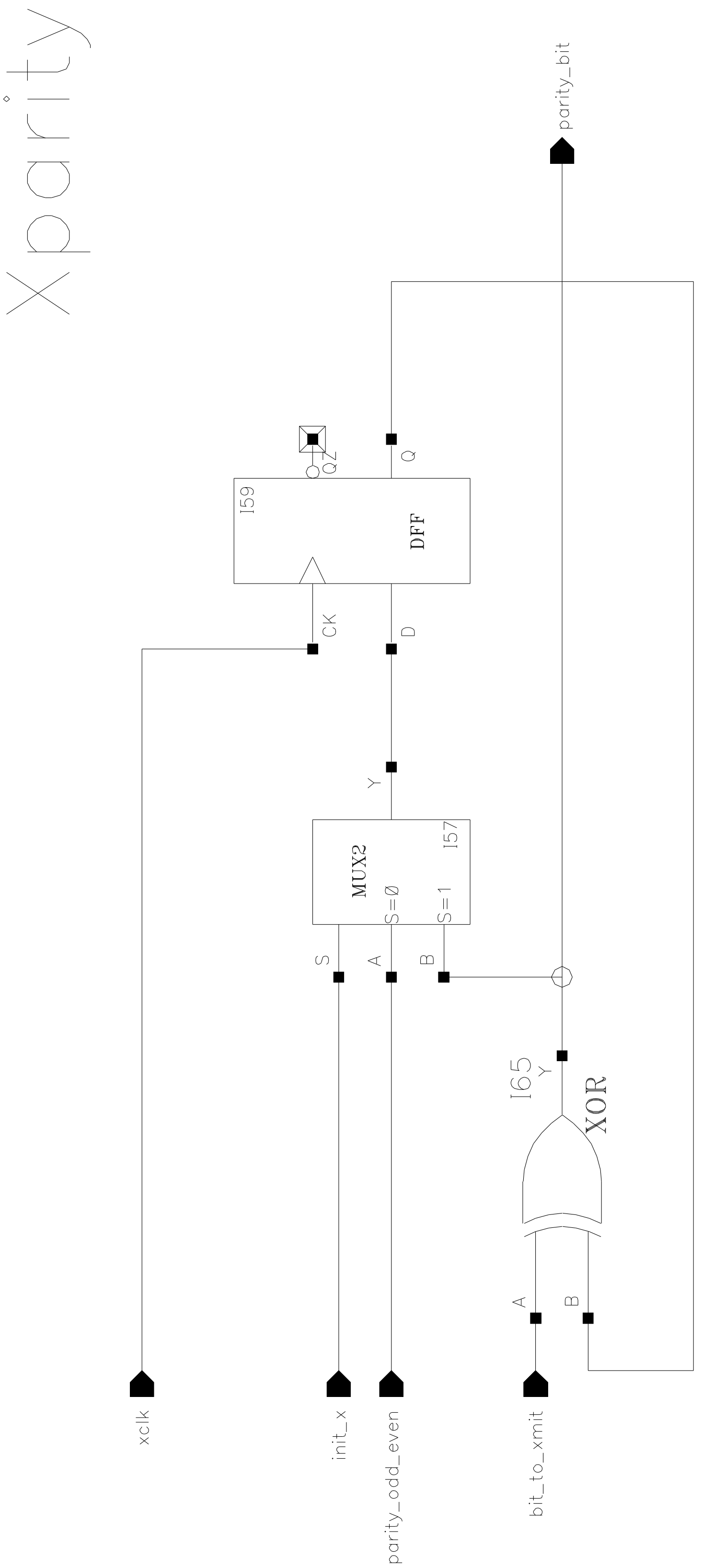


Figure A.3.1: Receiver General Block Diagram

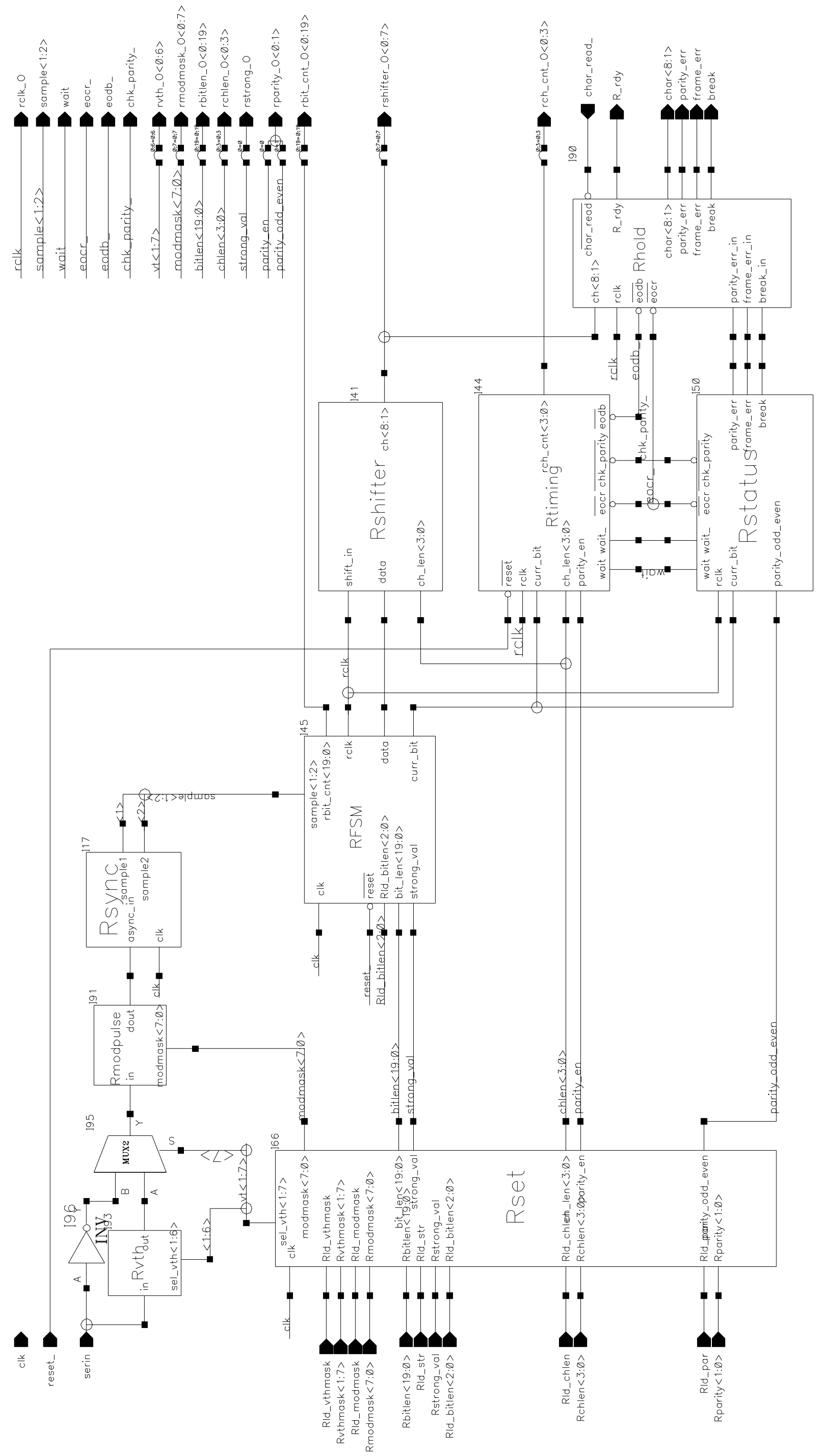


Figure A.3.2: Receiver Circuit for Setting of Parameters

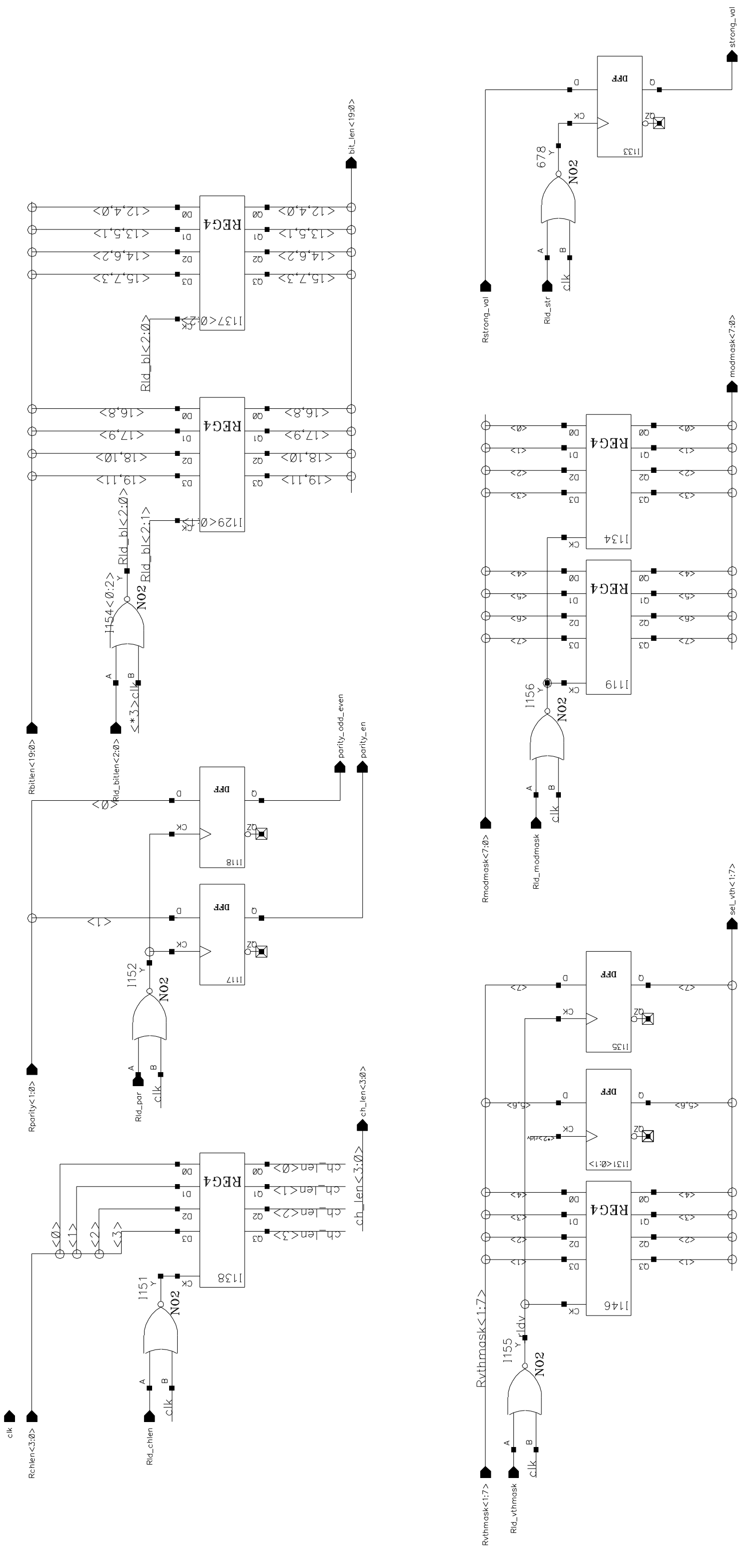


Figure A.3.3: Receiver Variable Input Threshold Inverter

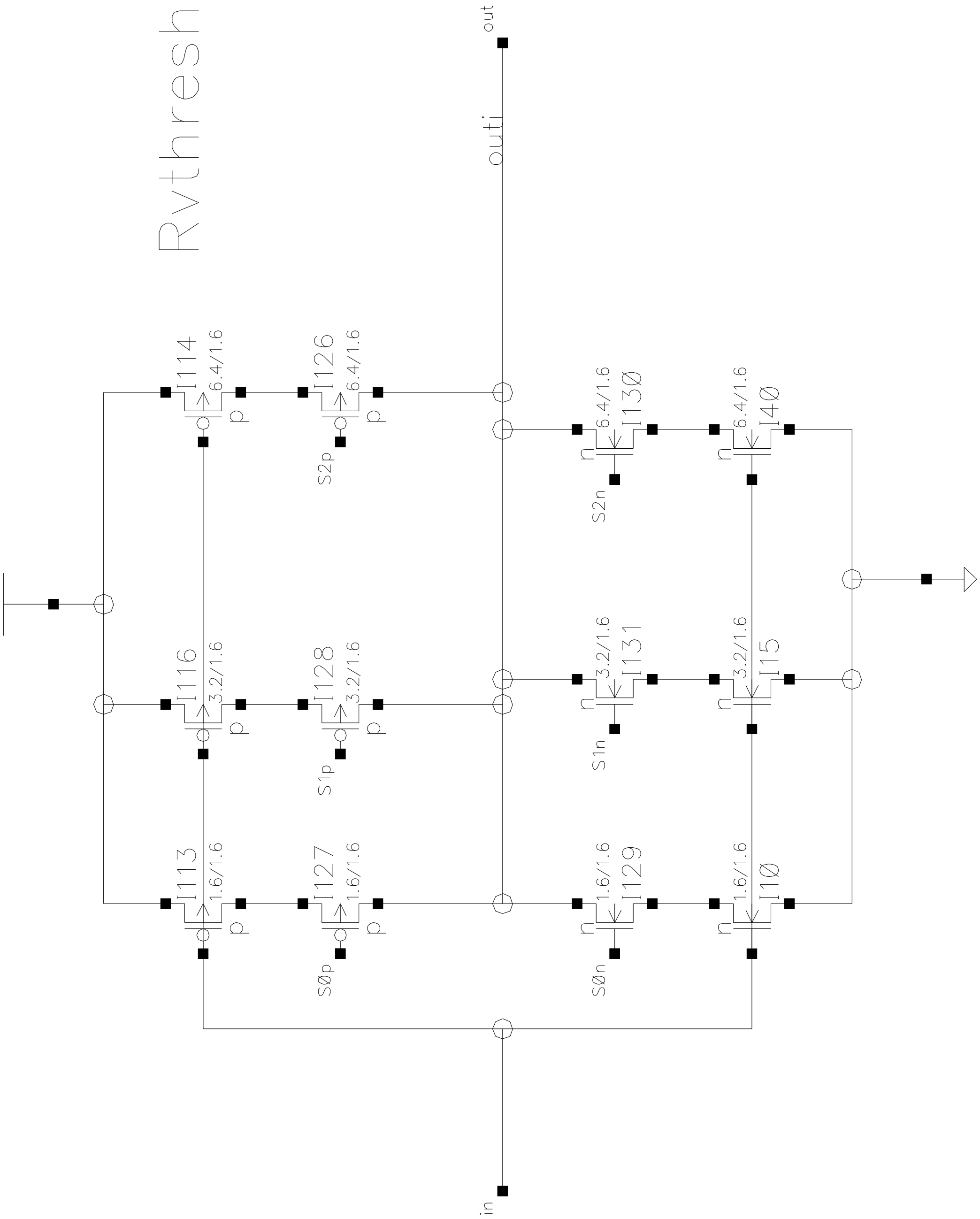


Figure A.3.4: Receiver Programmable Edge Delay Circuit

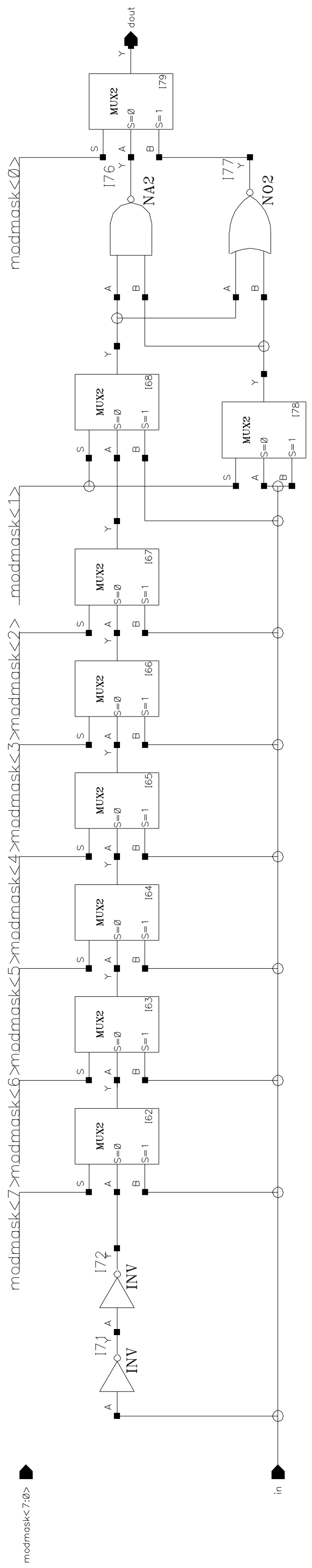


Figure A.3.5: Receiver Input Signal Synchronization Circuit

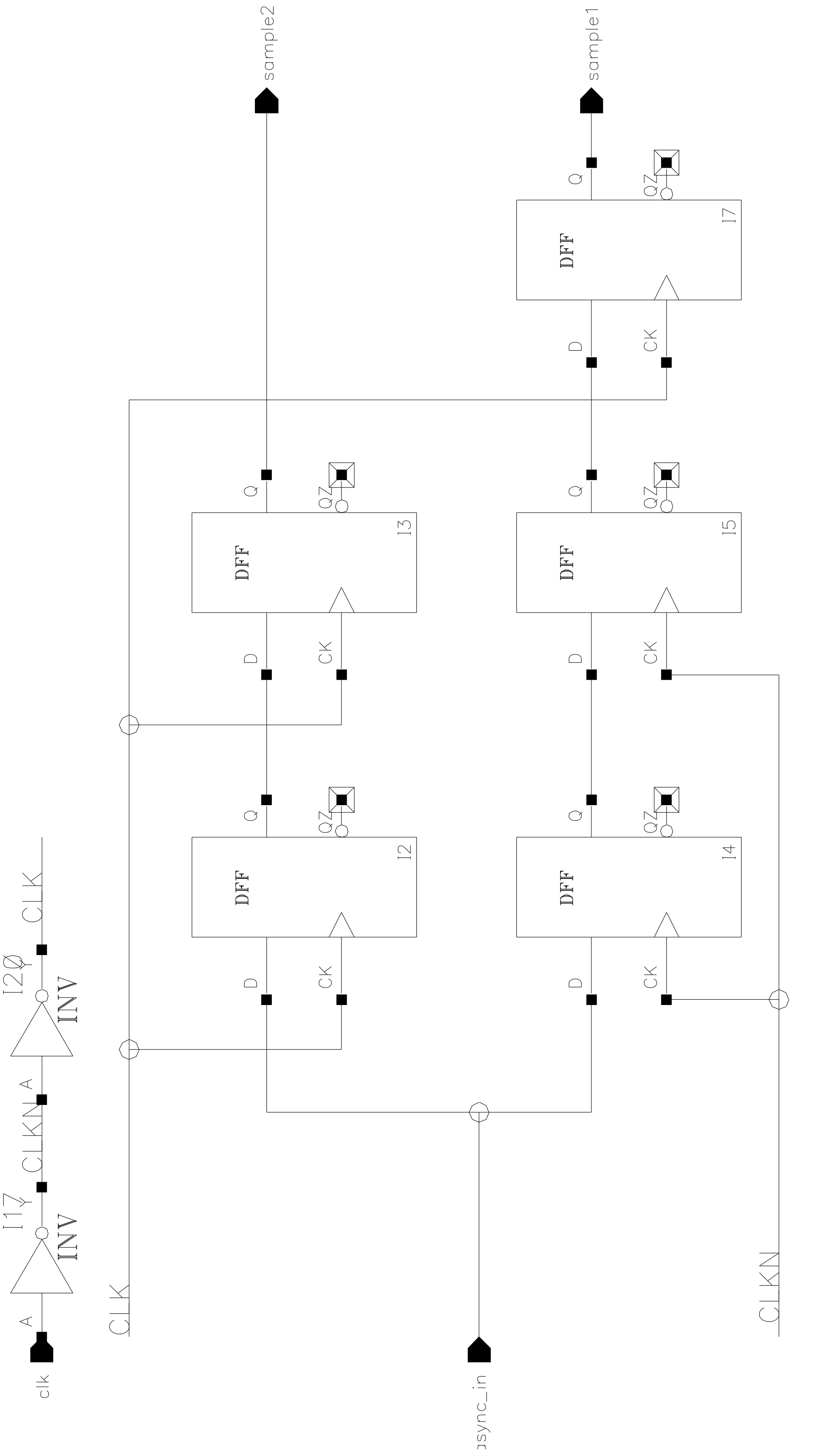


Figure A.3.6: Receiver Bit Finite State Machine

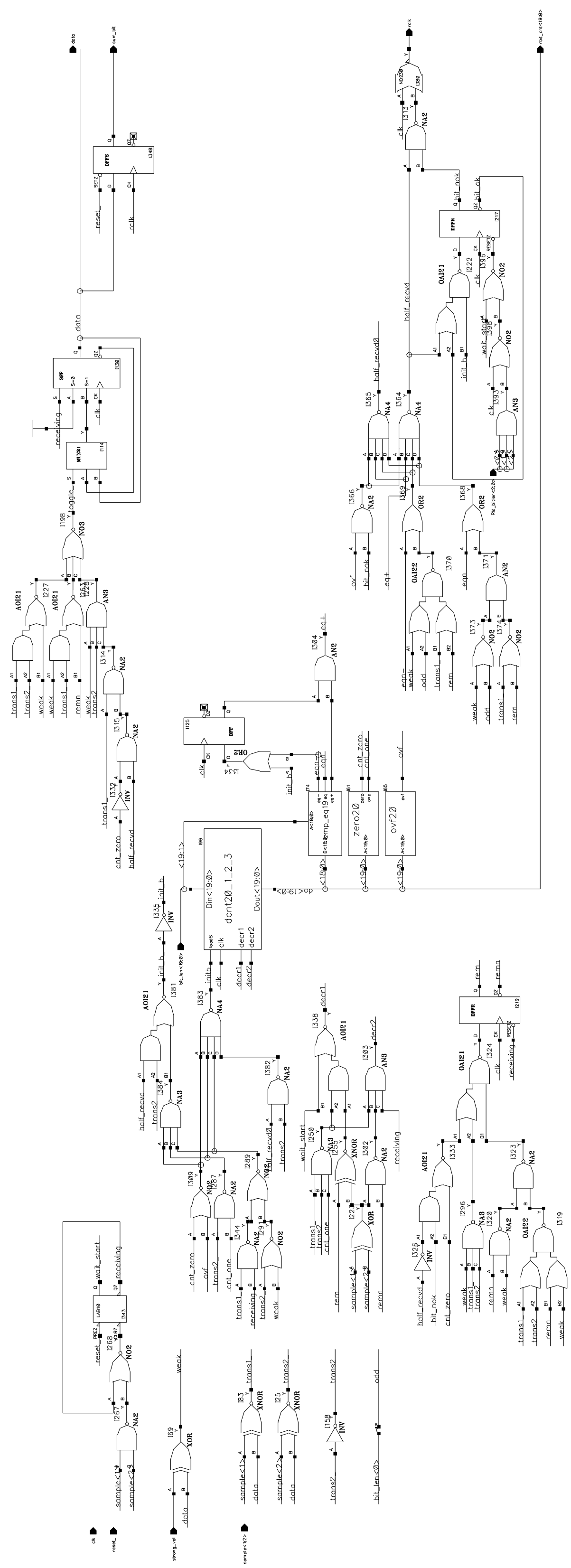


Figure A.3.7: Receiver Timing Signal Generation circuit

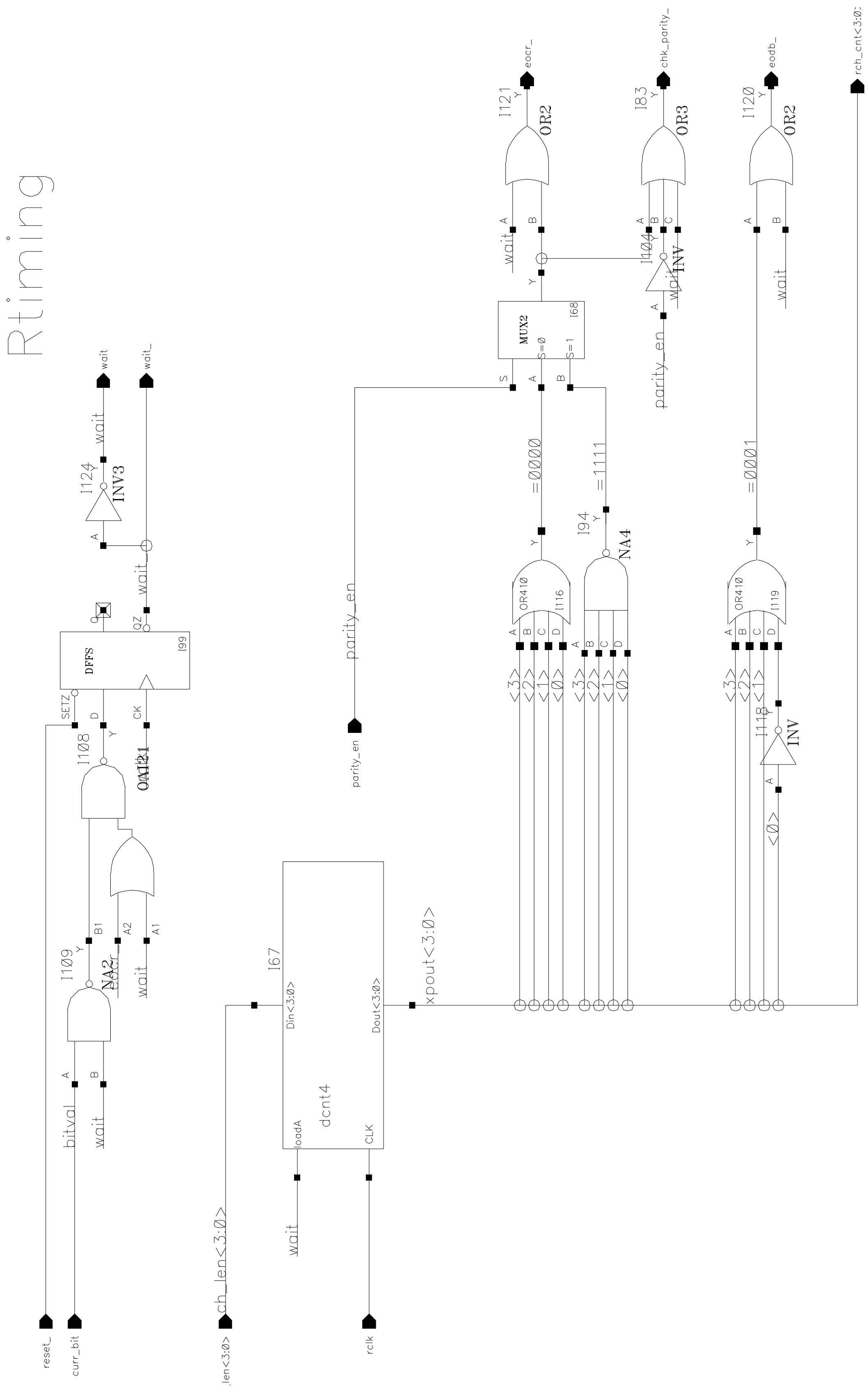


Figure A.3.8: Receiver Shifter Register

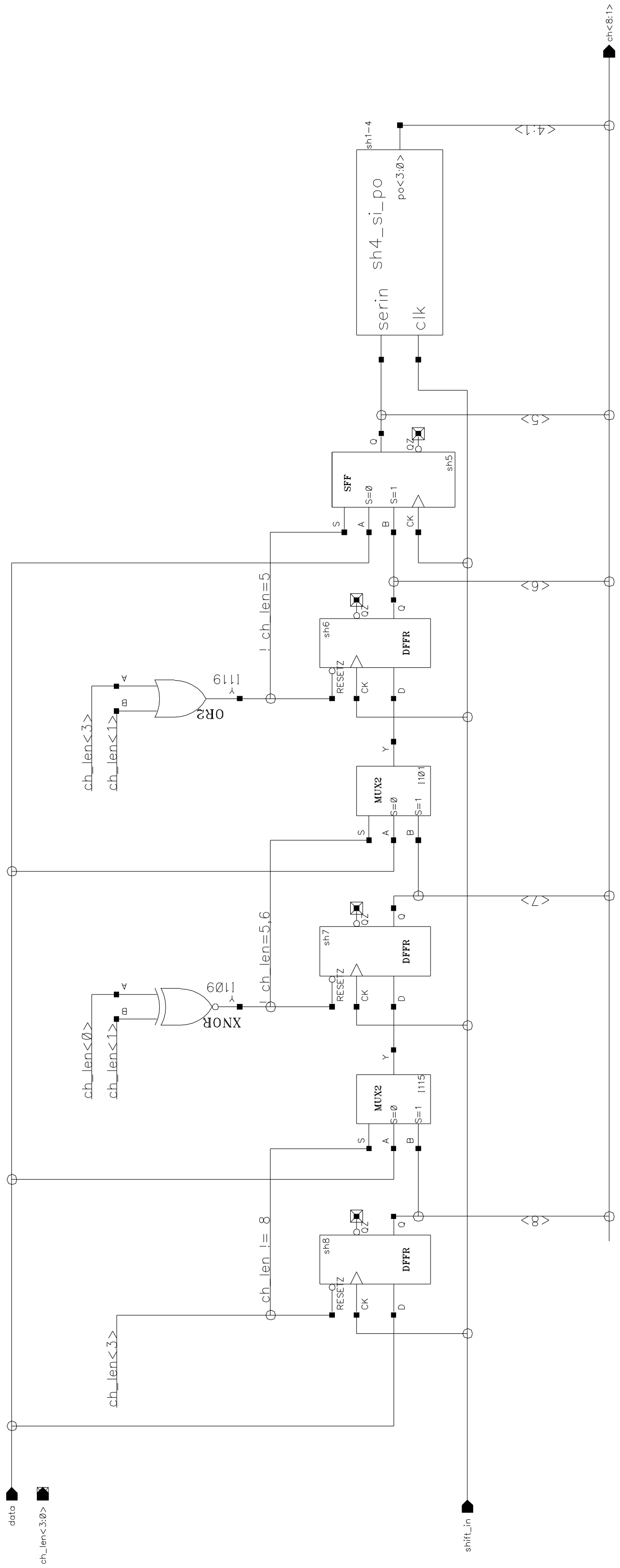


Figure A.3.9: Receiver Status and Error Cicruits

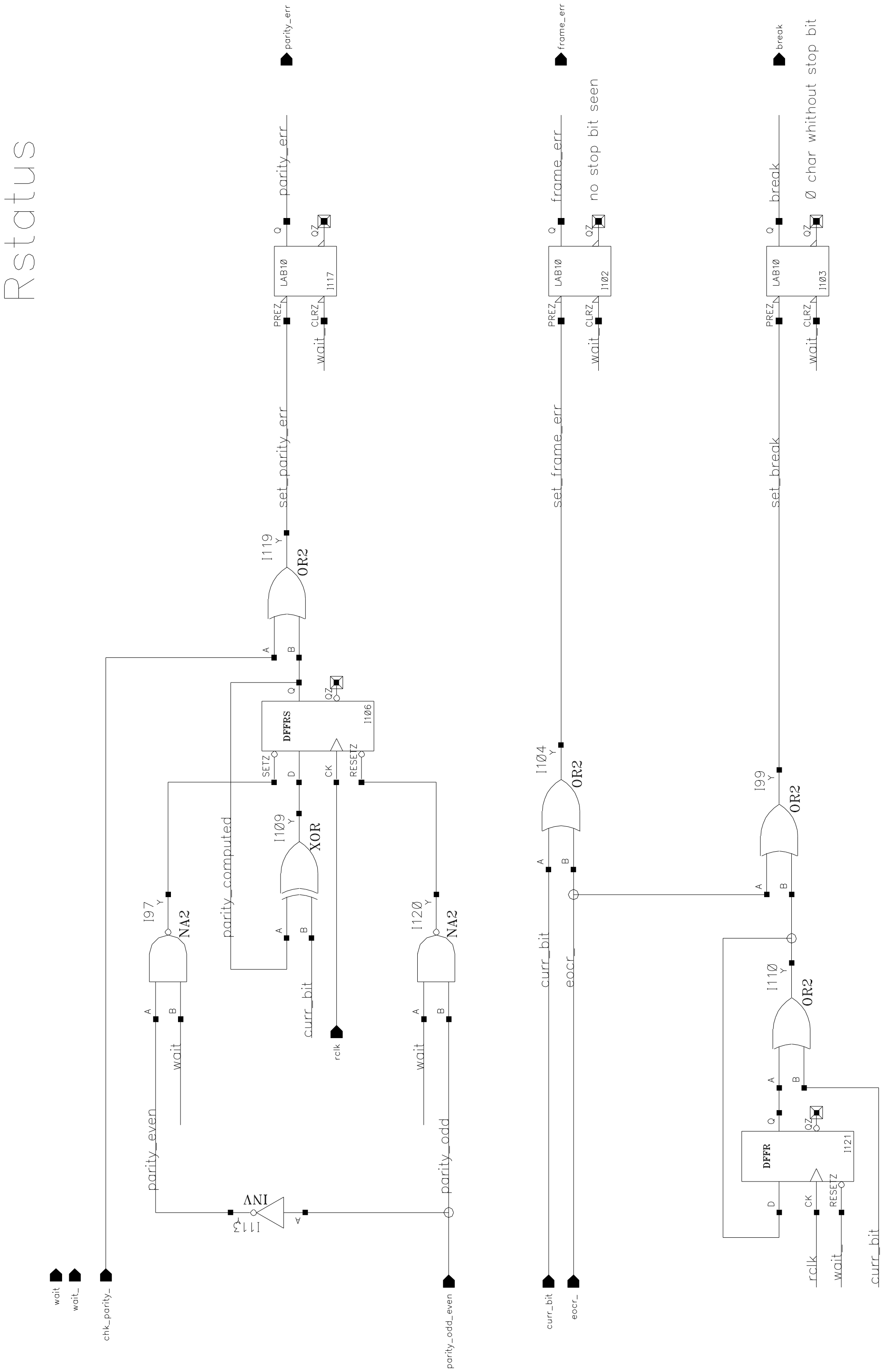


Figure A.3.10: Receiver Double buffering of Character and Status