

UNIVERSITY OF CALIFORNIA AT BERKELEY
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Checkpoint 1

N64 Controller

1.0 Motivation

This checkpoint serves two purposes:

1. Create the first and simplest module for your final project: a module to receive and process input from the N64 controllers.
2. Acquaint you with bit serial bus communications.

With the first goal in mind you should make sure to **design your verilog ahead of time**, comment you code and **test everything thoroughly**. Because you will be keeping and **relying on this code for months**, it will actually **save you many stressful hours to ensure it works well now**, rather than when you are about to finish the project.

Once upon a time it was too expensive to run parallel wires to form multi-bit busses as board and chip space were tight. Over time, as board space became less expensive and chips got bigger and more complicated, serial busses became too inefficient simply because they couldn't run fast enough, leading to a change to mostly parallel busses. In recent years however serial connections have become more important again with USB, 1GB Ethernet, SATA, PCI-Express, I²C and SPI not to mention the standard PS/2, RS232 and RS485 systems which have been around for decades. As such **learning to work with bit-serial protocols is a vital part of EECS150 and any digital design course**.

“BECAUSE YOU WILL BE KEEPING AND RELYING ON THIS CODE FOR MONTHS, IT WILL ACTUALLY SAVE YOU MANY STRESSFUL HOURS TO ENSURE IT WORKS WELL NOW, RATHER THAN WHEN YOU ARE ABOUT TO FINISH THE PROJECT”

2.0 Introduction

In this checkpoint you will be build what is at its heart **nothing more than a shift register**, or perhaps a pair of shift registers, implementing **a polling controller**. The sections below outline a more exact specification of how your circuit will need to operate, but the **basic operation sequence** is as follows:

1. **Wait for initialization** purposes
2. **Send a 8bit “Get Button Status”** command to the N64 Controller
3. **Receive a 32bit button status word**
4. Return to **step 2**

Of course this outline makes the checkpoint seem very simple, which is true from a functional standpoint. However for the first time, **guaranteeing timing will be a major part of your design and debugging time**.

2.1 Bidirectional Serial Protocols

Because the **interface between your circuit and the N64 controller is bidirectional**, bit serial and asynchronous, it presents several interesting design points that a normal point to point unidirectional bus does not. For example, **in a single source bus, there is no need for arbitration** because only one module ever drives the bus, whereas in a bi- or multi-source architecture, **there must be some scheme to prevent more than one module from driving the bus** simultaneously.

The most common solution, historically called an **open-collector bus**, is to use a **pullup resistor as shown in figure 1 below** to keep the serial data line high at all times and then **pull the line low to send a 1'b0 or leave it alone to send a 1'b1**. The only difference between this and a fully tristated bus is the pullup resistor used to ensure that the line is always driven by some source, and cannot be shorted.

As a mental exercise, you should **try work through why this bus design cannot result in a short circuit**.

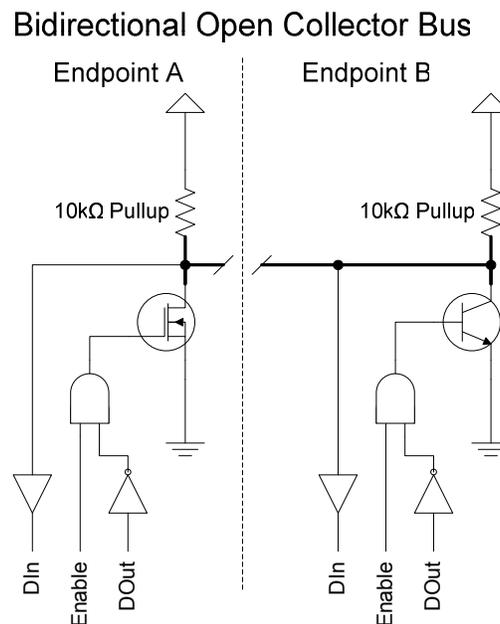


Figure 1: Generalized Bidirectional Open-Collector Bus Design

To implement this bus on the Xilinx chips, you will be using the tristate drivers built into the Virtex chips. **Shown in figure 2 below is the exact circuit you will need to implement in order to connect the N64 controller.** The entire circuit can be implemented using a **single assign statement**, if you keep in mind that **1'bZ is the verilog notation for tristating a wire**.

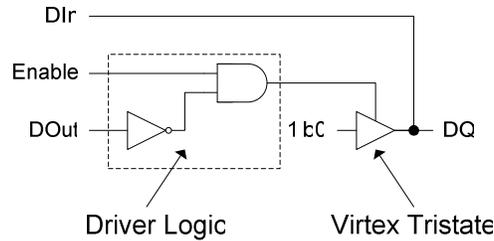


Figure 2: Virtex Open-Collector Driver

2.2 Asynchronous Serial Protocols

For many years the most common serial connection on computers was the asynchronous RS232 serial port running at 9600baud (bits per second). The main reason this connection survives even today is because it is electrically robust, requires only 3 wires and is completely asynchronous.

One of the primary reasons to use asynchronous serial protocols is to obviate the need for a clock signal to be transmitted along with the data. Because the clock line must switch at a minimum of twice the frequency of the data lines, it is much harder to transmit a clean clock signal. Even if the clock could be transmitted to the other end of the line, this would force the systems to be asymmetric as one would be sending and one receiving a clock, if we are to avoid any clock boundary crossing.

Using an asynchronous serial protocol means that there are no clock distribution problems, there is no power wasted in transmitting a clock, each node can be identical (they don't have to send or receive clock signals) and we can build a large system of identical nodes. The cost paid is in speed and receiver complexity.

2.2.1 Transmitting

Transmitting is simple, bits are send out at approximately the right rate along with parity or CRC error checking information, and some gaps between blocks of data. Because the protocol is asynchronous, we can just send bits out, and not worry about them; as long as the clocks at the two ends are “close enough” things will work out just fine.

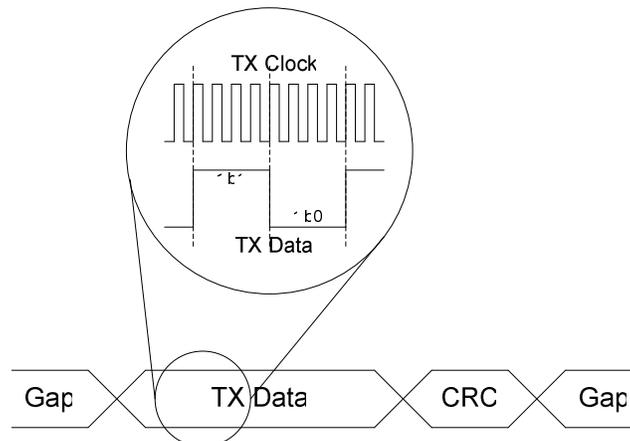


Figure 3: Asynchronous Serial Transmit

The CRC or parity information is used to check the data recovery in case the line is long, or the clocks are too far off.

The gaps between blocks of data are inserted in case the transmit speed is faster than the receive speed. If the clock tolerance of the protocol is $\pm 5\%$, then the bus must remain unused 10% of the time so that even if the transmitter is 5% fast and the receiver 5% slow, the receiver will only ever receive data at $100\% + 5\% + 5\% - 10\% = 100\%$ of the data rate.

Figure 3 above illustrates transmitting asynchronous serial data, showing the data, CRC and gaps between data. The detail also shows an important point: **the clock should be running faster than the actual data rate.** For transmit, this means that our output shift register will only shift on some cycles, but **it is vital for correct reception.**

2.2.2 Receiving

Given that the transmit side has ensured that data will be received at no more than 100% of the specified data rate (**accounting for clock frequency differences**), and has added parity or CRC information for integrity checking, the most complicated task of the receiver is **sampling the data line at the right times.**

Because the system clock might typically be 10MHz and the data rate as low as 100kHz, it is often quite easy to **oversample the incoming data to detect the boundary between bits** or bytes and then estimate when to sample using a counter. Essentially, **a counter is maintained which is reset every time a new bit or byte is detected** and then when the counter hits a specified value the data line is sampled and moved into a shift register.

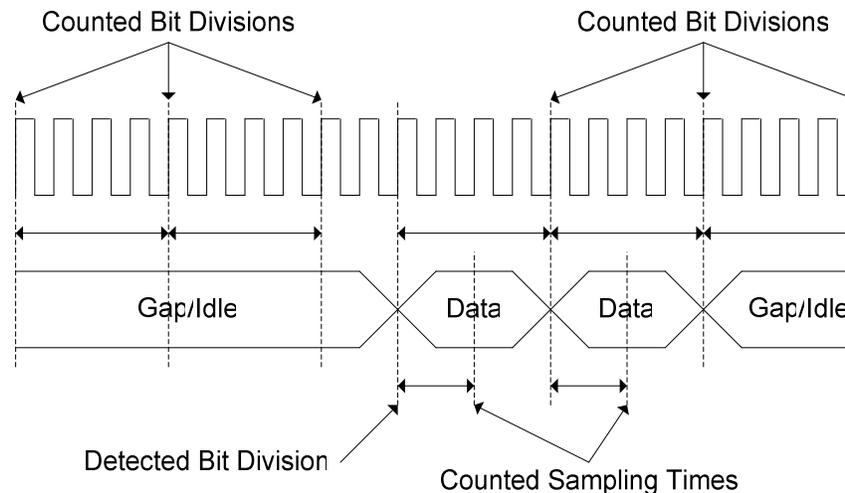


Figure 4: Asynchronous Serial Receive

Aside from **sampling bits at the right time**, the receiver must be able to **check the CRC** or parity information to ensure that the data hasn't been corrupted.

The receiver also needs to be able to detect situations where the line is remaining idle for a long time, even if another bit is expected. For example, if the cable is unplugged right in the middle of a data item, the receiver should not wait for the next bit forever, it must be able to "time out" and reset. We will discuss this more in section 2.4 N64 Serial Bit Timings.

2.3 N64 Commands

The N64 controller communicates with the CaLinx2+ board using a **single wire, bidirectional asynchronous serial interface**. Instead of having a complex arbitration scheme as mentioned in 2.1 Bidirectional Serial Protocols, the N64 controller **uses a challenge-response interface**: you will send the controller a command and it will respond.

There are 7 known commands for the N64 controllers:

- **8'hFF: Reset Controller**
- 8'h00: Get Status
- **8'h01: Get Buttons**
- 8'h02: Read Mempack
- 8'h03: Write Mempack
- 8'h04: Read EEPROM
- 8'h05: Write EEPROM

The two highlighted commands are those which you will be using.

The Reset Controller command you may optionally send whenever your circuit is reset. This will help the reliability of your circuit as it will ensure that the controller is properly reset.

The **Get Buttons command** is the primary command you will be using, **it will cause the controller to send back 32bits of button status information.**

Rather than include a CRC or parity bit along with each 8bit command or 32bit response, the designers of the N64 protocol chose to simply add a “**stop bit**” to indicate that the transmission is finished. Of course **since the stop bit is always 1'b1, it is impossible to really use this for error checking**, but you must still **transmit a stop bit** after every 8bit command and **be prepared to accept a stop bit after every 32bit response.**

2.4 N64 Serial Bit Timings

In addition to the stop bit after every command and response, the N64 protocol includes start and stop sub-bits to enable a receiver to detect when a new bit is coming in, as illustrated in figure 4, above. Figure 5, below, shows data transmission with the sub-bit system.

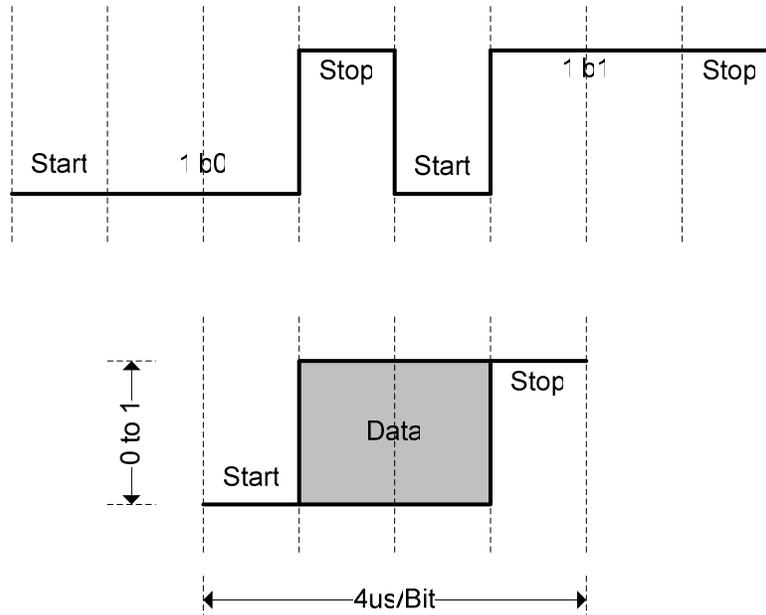


Figure 5: N64 Bit Timing with Start/Stop Sub-bits

The bottom of figure 5, shows the general format of each bit on the N64 data line, a 1'b0 start sub-bit followed by the actual bit data (which will be held for 2 sub-bits or 1us) followed by a 1'b1 stop sub-bit. The top half of figure 5 shows two bits being transmitted, a 1'b0 followed by a 1'b1, IE the word 2'h1 is being transmitted.

The most important point to notice in the above diagrams is that falling edges occur in exactly one place, right between the stop and start sub-bits. This provides an easy way to detect when a new bit is being received as detailed in section 2.2.2 Receiving and the lab lecture.

Shown in figure 6 below is an example of how to send the Get Button N64 command. This is the exact timing you must generate.

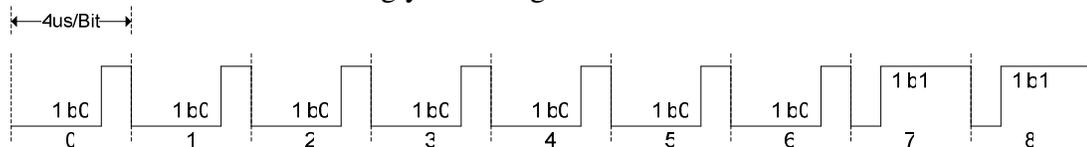


Figure 6: Get Button N64 Command (Exact waveform with stop bit and sub-bits)

3.0 Prelab

Please make sure to complete the prelab before you attend your lab section. **You will not be able to finish this lab in 3hrs otherwise!**

1. **Read this handout thoroughly.**
 - a. Pay particular attention to section **4.0 Lab Procedure** as it describes what you will be doing in detail.
2. **Examine the Verilog** provided for this weeks lab.
 - a. There isn't much, so it should be pretty clear.
3. **Write your Verilog ahead of time.**
 - a. **N64.v** will require significant debugging to finish. Make sure to write at least a draft of it ahead of time.

- b. **Attempt to create an N64Controller.v verilog model of the N64 controller which you can use to debug your N64.v file!**
 4. You will need the **entire 3hr lab!**
 - a. You will need to test and debug your verilog thoroughly.
 - b. **You must build a reliable interface with a real hardware component!**

4.0 Lab Procedure

Remember to **manage your Verilog, projects and folders well**. Doing a poor job of managing your files can cost you **hours of rewriting code**, if you accidentally delete your files.

4.1 N64.v

This is the **main module you will need to build** for this checkpoint. **At its core is a shift register** which is receiving the button status bits, **and a series of counters for generating all the bit and sub-bit timings** as well as the output Get Button command. The list below illustrates the sequence of operations that this module will need to go through while operating. Any time it receives a Reset, it should start again at the top of the list.

1. **Wait for initialization** purposes, you should wait at least 128 μ s
2. Optionally send a **“Reset” command** to the N64 Controller
3. **Send a 8bit “Get Button Status” command** to the N64 Controller
 - a. Be sure to include the **stop bit**
4. **Receive a 32bit button status word**
 - b. Be sure to accept the **stop bit**
5. **Wait a few microseconds**
 - c. This will help reliability and is optional
6. Return to **step 2**

Signal	Width	Dir	Description
Clock	1	I	The Clock signal
Reset	1	I	Reset all counters and registers (Not necessarily to 0)
N64_DQ	1	I/O	N64 serial data line (bidirectional, see figure 2)
DOut	30	O	Output status from the N64 controller
OutValid	1	O	Indicate that DOut is valid, should only be high 1 cycle for get time you get the button status.

Table 1: Port Specification for N64.v

Things to remember when building this module:

- You will need to **detect when a new bit is coming in** (see 2.2.2 Receiving)
- **The bus is bidirectional**, the schematic for the driver you will need to build is shown in figure 2 (see 2.1 Bidirectional Serial Protocols).

4.2 N64Controller.v

This module is not required to finish this checkpoint. All you need to do is build N64.v and demonstrate it working on the CaLinx2+ board. However you may find that being **able to simulate your N64.v module will make your life significantly easier**, this module will be a simulation model of the N64 controller with which your N64.v module must interact.

This module, **if you choose to build it, will be almost a direct copy of N64.v**, except that instead of sending an 8bit command it will receive one and instead of receiving 32bits of button status it will transmit those. **If your N64.v module is well written you may be able to copy it** and change no more than a small handful of lines to build this module

Signal	Width	Dir	Description
Clock	1	I	The Clock signal
Reset	1	I	Reset all counters and registers (Not necessarily to 0)
N64_DQ	1	I/O	N64 serial data line (bidirectional, see figure 2)
DIn	30	I	Output status from the N64 controller
InRequest	1	O	Indicate that DOut is valid, should only be high 1 cycle for get time you get the button status.

Table 2: Port Specification for N64Controller.v

4.2.1 Self Testing

Because your N64Controller.v module will likely be an (almost) copy of your N64.v module, there are some possible testing problems which may arise. For example you may inadvertently build a pair of matching bugs into the two modules which will allow them to work together, but will prevent the N64.v module from working with the actual N64 controller.

So why is this kind of testing useful?

Self testing can still help prevent a number of bugs, things from typos to un-driven signals to incorrect timing. Even though it is not as effective as testing against a known device model, this kind of self testing will allow you to see the behavior of your N64.v module over multiple Get Button request/response cycles.

We highly recommend that you attempt to simulate your module as it is the easiest way to verify that you got all the sub-bit timings and such correct, not to mention the fact that re-simulating after fixing a bug is much faster than having the synthesizer all over again.

5.0 Checkpoint1 Checkoff

Name: _____ Name: _____

Section: _____

- | | | |
|-----|--|-------------|
| I | Working N64 Module (Synthesis) | |
| | 1 Buttons | _____ (30%) |
| | 2 Analog Joystick | _____ (30%) |
| II | I/O Waveform | _____ (20%) |
| | 1 Simulation or Logic Analyzer | |
| III | Second Controller Test | _____ (20%) |
| | 1 Tested with more than one controller | |

IV Total: _____

V TA: _____

VI Hours Spent: _____

RevA - 9/22/2004	Greg Gibeling	...
------------------	---------------	-----