REAL TIME TRAFFIC SIGNAL INSTRUMENTATION USING NEMA TS2 SYNCHRONOUS DATA LINK CONTROL NETWORKS

A Thesis

Presented in Partial Fulfilment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Engineering

in the

College of Graduate Studies

University of Idaho

by

Jacob W. Preston

August 2014

Major Professor: Richard W. Wall, Ph.D.

Authorization to Submit Thesis

This thesis of Jacob W. Preston, submitted for the degree of Master of Science with a major in Computer Engineering and titled "Real Time Traffic Signal Instrumentation Using NEMA TS2 Synchronous Data Link Control Networks," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor _____Date_____
Richard W. Wall, Ph.D.

Committee Members _____Date_____
James F. Frenzel, Ph.D.

_____Date_____
Ahmed Abdel-Rahim, Ph.D.

Department Administrator _____Date_____
Fred Barlow, III, Ph.D.

Discipline's College Dean _____Date_____
Larry Stauffer, Ph.D.

Final Approval and Acceptance by the College of Graduate Studies

_____Date_____
Jie Chen, Ph.D.

Abstract

In this thesis, a method of real time traffic signal instrumentation is introduced and described. Current practices of cabinet instrumentation methods require physical electrical connections to each traffic signal being instrumented. These methods are labor intensive to install and some state governments require a certified technician due to the 120VAC signals used. The new method obtains information concerning the traffic signal status by monitoring communications on the Synchronous Data Link Control (SDLC) network between existing equipment within NEMA TS2 traffic cabinets. Using the SDLC interface reduces the risk associated with high voltage and allows for a more time efficient installation.

A case study is presented that demonstrates the proposed SDLC–based instrumentation method with an Advanced Accessible Pedestrian System (AAPS). Comparisons between the proposed SDLC method and current methods of traffic signal instrumentation showed that the decoding time of sensing a traffic signal state in the new method is more consistent with devices already in the traffic cabinet. This new method can be used in other traffic signal system applications such as an SDLC Interface Device or a device to log the state of different input/output channels at the signalized intersection.

Acknowledgments

First and foremost, I would like to thank my major professor, Dr. Richard Wall, for his continued support of my work and education. Without him, I wouldn't have been presented with the wonderful opportunities to expand my skills and expertise that I have so fortunately been granted.

I would like to thank those at Campbell Company in Boise, ID for sponsoring this project: Phil Tate, Zane Sapp, and Cody Brown for helping me get things started.

A large part of this work relied on my HDL skills, which would be non-existent if it weren't for Dr. Jim Frenzel and his wonderful teaching methods to keep me motivated in my coursework.

Ryan Kalis in the University Writing Center was a great help in identifying parts of my writing that could have been better.

Finally, I would like to thank Dr. Mike Dixon and Dr. Ahmed Abdel-Rahim for giving me the opportunity to continue my work and further advance my skills. They were both constantly open to my ideas.

Table of Contents

List of Acronyms

AAPS . . . . Advanced Accessible Pedestrian System

AC . . . . . Alternating Current

ACK . . . . . Acknowledge

APB . . . . . Advanced Pedestrian Button

APC . . . . . Advanced Pedestrian Coordinator

APS . . . . . Accessible Pedestrian Signal

ASIC . . . . . Application Specific Integrated Circuit

ATC . . . . . Advanced Transportation Controller

BIU . . . . . Bus Interface Unit

CA . . . . . Controller Assembly

CID . . . . . Controller Interface Device

CPLD . . . . Complex Programmable Logic Device

CRC . . . . . Cyclic Redundancy Check

CU . . . . . Controller Unit

DC . . . . . Direct Current

DCO . . . . . Digitally Controlled Oscillator

DR . . . . . Detector Rack

DW . . . . . Don't Walk

EFB . . . . . Embedded Function Block

EOP . . . . . Ethernet Over Powerline

FCS . . . . . Frame Check Sequence

FDW . . . . . Flashing Don't Walk

HDL . . . . . Hardware Description Language

HIL . . . . . Hardware-in-the-Loop

I2C . . . . . Inter-integrated Circuit

LED . . . . . Light Emitting Diode

MMU . . . . Malfunction Management Unit

MUTCD . . . Manual on Uniform Traffic Control Devices

NACK . . . . Not Acknowledge

NEMA . . . . National Electrical Manufacturers Association

NIATT . . . . National Institute for Advanced Transportation Technology

PCB . . . . . Printed Circuit Board

PFU . . . . . Programmable Function Unit

RITA . . . . . Research and Innovative Technology Administration

RMS . . . . . Root Mean Square

SBC . . . . . Single Board Computer

SDLC . . . . Synchronous Data Link Control

SoC . . . . . System-on-Chip

SPI . . . . . Serial Peripheral Interface

TF . . . . . Terminals and Facilities

VAC . . . . . Volts Alternating Current

VHDL . . . . VHSIC Hardware Description Language

VHSIC . . . . Very High Speed Integrated Circuit

W . . . . . Walk

List of Figures

List of Tables

List of Listings

Chapter 1: Introduction

Over 320 billion vehicle trips and over 40 billion walking trips were made in the United States in 2009 [1]. With such a large amount of vehicle and pedestrian traffic, technology must be used in order to keep traffic flow organized. Signalized intersections are one approach to coordinate traffic flow. Signalized intersections are designed to promote safety and efficiency of traffic flow [2]. In order to design a signalized intersection in this way, researchers are looking for new methods to gather information regarding traffic flow and the use of signalized intersections. This thesis discusses an approach to retrieve intersection signal state information within a traffic cabinet. It is also shown that the same methods for retrieving intersection information can be applied to technologies such as Accessible Pedestrian Signals (APS) and Hardware-in-the-Loop (HIL) traffic simulation devices.

1.1 Overview of Signalized Intersections

Signalized intersections direct right of way of conflicting traffic movements [3]. Right of way is indicated to each approach to the intersection via different signal lights. Vehicle signals are composed of three different colored lights (red, yellow, and green) to specify if an approach currently has the right of way. Red lights are used to signal to the driver that they are not permitted to enter the intersection. Green lights are used to specify that the approach has the right of way. Yellow lights are used to indicate that the light is about to turn red. Each vehicle approach to the intersection will have its own set of signal lights. A detailed description of each signal light color and its function can be found in the Manual on Uniform Traffic Control Devices (MUTCD) 2009 [2].

Figure 1.1 shows an example of an eight-way intersection. Each of the eight approaches are assigned a specific movement phase. These phases are used by a Controller Unit (CU), also known as a traffic controller, to coordinate and permit non-conflicting phases where a phase is a direction of travel through the intersection. For example,

Phases 8 and 3 can be permitted at the same time. Phases 8 and 6 cannot be permitted at the same time. The CU must ensure that conflicting phases are not permitted simultaneously.

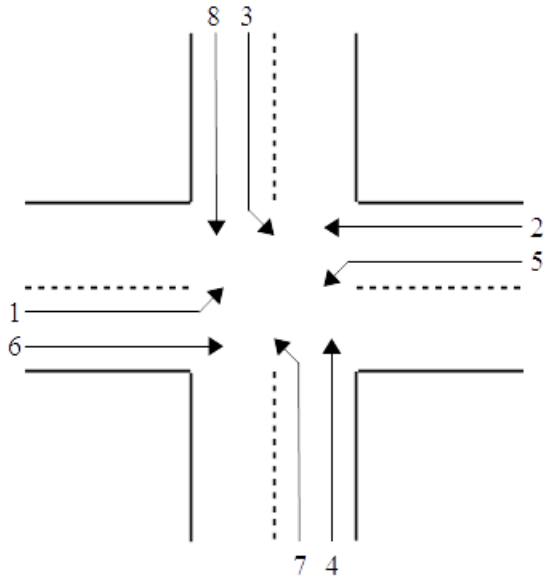

Figure 1.1: Phase description of an eight-way intersection.

Pedestrian right of way is managed similarly to vehicles. Pedestrian signals contain two signal lights represented by a walking man and a red hand. The walking man (walk sign) indicates that the pedestrian is permitted to enter the intersection. The red hand (don't walk sign) can be in one of two states: flashing or solid. If the red hand is solid, the pedestrian approach is in the "Don't Walk" state and the pedestrian should wait for the next time the walk sign is on. If the red hand is flashing, the pedestrian approach is in the clearance interval state. The clearance interval is intended to allow pedestrians that have already started crossing the intersection to complete crossing [2].

For a signalized intersection to effectively manage right of way, it must recognize when vehicles or pedestrians are present. There are multiple vehicle detection methods such as video cameras and inductive loops [4]. Video cameras operate by detecting a change in contrast when a vehicle approaches the intersection. Inductive loops are embedded in

the roadway surface and have an electric current flowing through them. When a vehicle passes over an inductive loop, the frequency of the current flow will fluctuate due to the metallic material of the vehicle. This fluctuation allows the signalized intersection to determine that a vehicle is present.

Pedestrian detection is typically done with a button. Buttons are installed at the corners of the intersection that pedestrians may cross. Pressing the button will express to the intersection that a pedestrian is present. Other methods of pedestrian detection have been investigated such as video cameras [5] and infrared sensors [6], among others.

Some intersections operate using a fixed-time scheme [7]. That is, each approach will get a certain amount of time to enter the intersection regardless of how many vehicles or pedestrians are queued. Fixed-time intersections do not detect the presence of vehicles or pedestrians. Some intersections implement an exclusive pedestrian walk phase where only pedestrians are permitted to enter the intersection. These are typically used in large cities with a large amount of pedestrian traffic [8].

1.2 Pedestrian Systems Overview

Pedestrian systems come in many varieties. Some pedestrian call buttons are mechanical buttons that give no visual or audio feedback when pressed. Other pedestrian buttons will beep and turn on a Light Emitting Diode (LED). There is currently no way to confirm that the traffic controller recognizes the button has been pressed. Red LEDs and beeping is currently done at the button and merely indicates that the button has been pressed but gives no feedback from the traffic controller. More advanced pedestrian systems will give verbal feedback to confirm a button press. Pedestrian systems can also relay important information regarding the state of the pedestrian signal via a variety of chirps and beeps, verbal messages, or vibrotactile pulses.

The one thing all pedestrian call buttons have in common among popular signalized intersection standards is the method used to place a call to the CU. From the CU's

perspective, a pedestrian call is a simple binary input: either the button is pressed or it isn't. Thus, the only information that will alert a CU to the presence of a pedestrian is the pedestrian call and the length of time the pedestrian had the call button pressed.

1.2.1 Accessible Pedestrian Signals

The earliest known pedestrian signals that included more than a visual signal were in use in the United States in the 1920s [9]. These systems were typically installed near schools for the visually disabled. Even though this technology has existed for quite some time, APS didn't have a strict definition until the publication of the MUTCD 2000 [2].

Pedestrian buttons without feedback can give the pedestrian little confidence that the intersection has actually received the pedestrian's request. This can incorrectly motivate the pedestrian crossing the intersection before a walk sign turns on. Furthermore, the only indication the pedestrian will receive when the request is serviced by a traffic controller is the inaudible change of a signal light. Rather than being a pure visual signal like many pedestrian crossings at signalized intersections today, APSs give pedestrians useful information regarding the state of the pedestrian signal via multiple sensory inputs. The MUTCD 2009 defines APSs as "a device that communicates information about pedestrian signal timing in non-visual format such as audible tones, speech messages, and/or vibrating surfaces" [2].

R.V. Houten *et al* have shown that pedestrian signals that give more than a visual signal increase pedestrian use at call buttons as well as increasing the likelihood that a pedestrian will wait for the signal to change [10]. Therefore, it can be inferred that pedestrian call buttons that confirm when they are pressed create a safer environment for pedestrian use of signalized intersections.

1.3 Background and Motivation

The signalized intersection is a key element in providing safe and efficient use of right of way shared between vehicular and pedestrian modes of transportation. When compared to a passenger vehicle which typically incorporates safety features such as air bags, seat belts, and crumple zones, the pedestrian is the more vulnerable user of the traffic intersection. A pedestrian will rely on sight and sound to alert him or her to danger. The 2010 National Health Interview Survey reports 21.5 million Americans aged 18 or older are either blind, or claim that they have trouble seeing, even when wearing corrective lenses [11]. This leads one to conclude that a large number of pedestrians have difficulty crossing a signalized intersection confidently due to the challenge of deciphering the signal light based on vision alone. Even a pedestrian with perfect vision can have his or her view of a pedestrian signal blocked by a vehicle, poor weather conditions, other pedestrians, poor placement of the signal, or other distractions. Figure 1.2 shows one example of how distracting an intersection can be by giving drivers and pedestrians conflicting information.

Figure 1.2: Example of an intersection presenting conflicting information [29].

Pedestrians rely on other senses to gain information regarding the intersection when their view of the signal light is blocked or obscured. Therefore, it is imperative that the pedestrian system at a signalized intersection give more than just a visual signal to assist in the pedestrian's decision to start crossing. APS systems will give the pedestrian an audible tone when a change is detected in the pedestrian signal. APSs must also include vibrotactile responses to relay information via the sense of touch.

In 2004, Richard Wall began research and development of using distributed signal technology with traffic signals at the University of Idaho in Moscow, ID [12]–[14]. The proposed method of utilizing distributed systems at a signalized intersection allows the application of plug and play concepts to an APS system. Distributed computing enables the dispersion of intelligent computation throughout the intersection. In contrast, many signalized intersections contain all computation within the traffic cabinet and utilize long wires routed throughout the intersection to control signal lights and detection devices.

1.3.1 Advanced Accessible Pedestrian System

The system designed by Richard Wall and other researchers is known as an Advanced Accessible Pedestrian System (AAPS). By using distributed systems design, the Advanced Pedestrian Coordinator (APC)–which is the central pedestrian controller within a traffic cabinet, may communicate pedestrian signal information to each pedestrian button distributed throughout the intersection. Each Advanced Pedestrian Button (APB) is able to output audible messages and vibrotactile responses depending on the state of the corresponding pedestrian signal. The state of the pedestrian signal is determined by the APC which then communicates this information to all APBs in the intersection.

The audible messages AAPSs include are locater tones, wait messages, and verbal information messages. Locater tones are a simple repetitive tone that allow a pedestrian to locate the button. Wait messages inform the pedestrian to wait to cross the intersection after the pedestrian has pushed the call button on the APB. The wait message

may include additional information about the intersection such as street names. Other verbal messages include indicating which street the walk sign has just turned on for.

The wait message may depend on the duration of the button press. A standard short press of the button causes the APB to output a simple wait message–usually the word "Wait". An extended press has the option of giving the pedestrian more information as described in section 4E.13 of the MUTCD 2009 [2]. The MUTCD 2009 defines a standard short press as less than one second and an extended press as greater than or equal to one second. The extended press can be used to give pedestrians more time to cross the intersection if the traffic controller and APC are configured to do so. According to the MUTCD 2009, the special functions for an extended press are not required for a pedestrian system to be considered an APS.

Ordinarily, a traffic controller will only receive an electric signal indicating that the button has been pressed. This is referred to as a pedestrian call. The only information the traffic controller knows about the pedestrian is which street corner the call was placed on. It is possible that the traffic controller has been enabled to increase the duration of the pedestrian clearance interval (indicated by a flashing don't walk sign) if an extended press is used. However, this function is rarely enabled [15]. Since the traffic controller receives little information regarding the pedestrian, it is crucial that an AAPS be able to sense pedestrian signals accurately and confidently using the pedestrian system's own method in order to determine which audible messages to play at each APB.

1.4 Thesis Organization

This thesis presents a method of real time traffic signal instrumentation utilizing the Synchronous Data Link Control (SDLC) network within NEMA TS2 traffic cabinets. This method can be utilized by AAPSs. For example, the APC will determine the state of pedestrian signals by deciphering messages transmitted on the SDLC network. This method can be used in other applications such as HIL simulation devices.

The organization of this thesis is as follows: Chapter 1 introduces signalized intersections and APS systems. Chapter 2 gives an overview of modern day traffic control, particularly the NEMA TS2 standard. Chapter 3 describes the NEMA TS2 SDLC network in detail and the advantages of using it for traffic signal detection. Chapter 4 describes the hardware designed to decode traffic signal state from the SDLC network using a Complex Programmable Logic Device (CPLD). Chapter 5 demonstrates an AAPS implemented with the SDLC method of traffic signal instrumentation. Chapter 6 presents the results of the AAPS using the SDLC network. Chapter 7 concludes the thesis and describes future applications that can use the same method shown in the AAPS such as a Controller Interface Device (CID) for HIL simulation of an intersection. Appendix A shows a schematic of the new APC for an AAPS. Appendix B shows the Hardware Description Language (HDL) code for the CPLD.

Chapter 2: Traffic Control Technologies

There are multiple traffic control standards for signalized intersections. Today's most up-to-date standards utilize solid state technology to control the signal lights at an intersection. Among today's standards are NEMA TS1 [16], NEMA TS2 [17], Type 170 [18], and ATC Type 2070 [19]. Some older signalized intersections may utilize electro-mechanical designs within a traffic controller to directly control signal lights. For the most part, the electro-mechanical controller has been phased out by solid state controllers.

A survey was performed by the Research and Innovative Technology Administration (RITA) to gather statistics on the deployment of ITS technology throughout the United States in 2010 [20]. Almost three hundred traffic agencies responded to the survey, and Table 2.1 summarizes their responses (note that some agencies are using more than one technology). As shown by the table, NEMA TS2 is a popular technology with about 44% of the surveyed agencies implementing NEMA TS2 in at least some of their signalized intersections. The total number of NEMA TS2 implementations reported by the survey is 29,058.

Technology	**Agencies**	**Controllers Deployed**
NEMA TS2	129	29,058
Other	114	Not Listed
Type 2070	83	16,668
Type 170	72	26,826

Table 2.1: Summary of the number of agencies deploying different signalized intersection control technology out of 290 surveyed agencies.

2.1 NEMA Standards

NEMA TS1 was the first NEMA standard to implement solid state technology for the purposes of traffic control and was first defined in 1976 [16]. The limitations of the

TS1 standard include overly abundant wiring, lack of uniform implementation, and the out of date technology used. The NEMA TS2 standard was created and approved in 1992 in order to address these limitations in the NEMA TS1 standard. Although based on the NEMA TS1 standard, NEMA TS2 adds stricter definitions and optional features [17], [21]. Since its inception, the NEMA TS2 standard has been revised and modified to meet the requirements of present day effective and safe signalized traffic control.

The goals of the NEMA TS2 standard are to create a means of traffic control that is interchangeable among different implementations of the same standard, minimize the potential for equipment malfunctions, and provide future expandability, among others. See the "History" section in [17] for a full description of the motivation behind the creation of the NEMA TS2 standard.

To maintain backwards compatibility with NEMA TS1 implementations, the NEMA TS2 standard defines two different types of CUs: Type 1 CUs are for a pure NEMA TS2 Controller Assembly (CA), while Type 2 CUs are meant for upgrading a NEMA TS1 CA but don't require that the other components within the cabinet be upgraded to the NEMA TS2 standard. NEMA TS2 Type 2 CUs have all the connections necessary to be implemented in a NEMA TS1 CA. This includes all the individual connections that control individual signal lights in circular connectors labeled A, B, and C. These circular connectors include inputs from detectors, pedestrian buttons, and so on. Some traffic controllers include a D connector for the manufacturer's use, which is not defined in the NEMA TS1 standard.

The NEMA TS2 standard replaces the A, B, and C connectors with a digital serial interface bus to communicate with all devices within the cabinet. This is an SDLC bus defined by IBM [22]. For TS2 Type 2 cabinets, the only devices required to be connected to the SDLC bus are the CU and Malfunction Management Unit (MMU). All other devices will be connected to the CU via the old-style A, B, and C connectors. For TS2 Type 1 cabinets, the CU and MMU along with Bus Interface Units (BIUs) are

required to be connected to the SDLC bus. This document discusses devices that utilize the SDLC network within NEMA TS2 traffic cabinets. Because of this, these devices can only be installed in NEMA TS2 cabinets.

2.1.1 NEMA TS2 Devices

NEMA TS2 traffic cabinets are composed of a few essential components: the CU, the MMU, and a variable number of BIUs (see Figure 2.1 for a block diagram of a NEMA TS2 – Type 1 traffic cabinet where N specifies the number of signal lights). There are two types of BIUs defined: Terminals and Facilities (TF) BIUs and Detector Rack (DR) BIUs. The number of BIUs utilized in a NEMA TS2 cabinet depends on the number of phases and implementation of the intersection. The NEMA TS2 standard allows for up to 16 BIUs, but only four TF BIUs and four DR BIUs have explicitly defined functionality. This gives way to future functions that may be required in subsequent revisions of the NEMA TS2 standard.

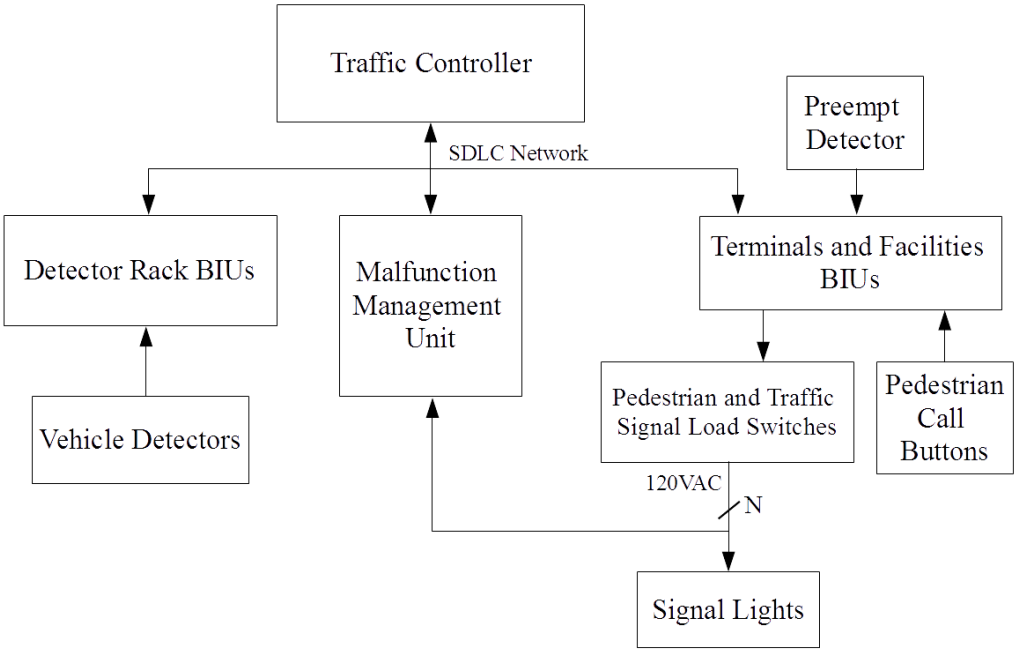

Figure 2.1: NEMA TS2 – Type 1 traffic cabinet block diagram.

The CU is the main traffic controller. It runs algorithms and timing schemes to

control the state of signal lights throughout the intersection by reacting to vehicle, pedestrian, and preempt detections. The MMU monitors the intersection for conflicting signals. The MMU detects conflicting signals by monitoring the outputs of load switches that control signal lights. If the MMU detects a conflict, it will notify the CU and cause the intersection to go into a flash mode of operation where all approaches will go into either a flashing yellow state or a flashing red state. Typically, intersections will flash red at all approaches when the MMU detects a conflict [23]. DR devices are connected to vehicle detection devices such as video cameras or inductive loop sensors. TF devices do multiple things: control the voltage to traffic signal lights by toggling load switch outputs, detect pedestrian calls, and detect preemption calls. Preemptions are special calls placed to the CU such as railway preemptions, emergency vehicle preemptions, and public transit preemptions. Preemptions override the current timing scheme of the traffic controller and force the intersection into a state that permits the vehicle that caused the preemption.

The CU acts as the primary device on the SDLC bus. The traffic controller communicates to all other devices on the bus to control the state of the intersection. BIUs give TF devices and DR devices an interface to the traffic controller via the SDLC bus. Up to 16 vehicle detectors can be connected to a single DR BIU. The MMU is used to monitor physical traffic signals via their 120VAC inputs from load switches and ensures that no conflicting signals occur based on jumper configurations on the programming interface card [24]. Some examples of possible intersection operations that would cause the MMU to put the intersection into a flash mode of operation are conflicting green lights and burnt out signal lights that give a false indication that a signal is turned on.

2.2 Pedestrian Systems

As an application for pedestrian systems is discussed in this thesis, it is important to have an overview of pedestrian systems in general. As stated previously, the CU

receives little information regarding pedestrians. However, the pedestrian system itself can perform its own computation in order to add functionality to the pedestrian system while still maintaining compatibility with current traffic control technologies.

2.2.1 AAPS Hardware

One such system for adding functionality to pedestrian systems is the AAPS designed at the University of Idaho and currently being manufactured by Campbell Company in Boise, ID [25]. The two major components of an AAPS are the APC and the APB. APCs are located within the traffic cabinet and are connected directly to the 120VAC load switches that control pedestrian signals. An APC senses what state a pedestrian signal is in using optocouplers connected to a microprocessor. Optocouplers electrically isolate two sides of a circuit with different voltage sources and voltage levels. Figure 2.2 shows an example circuit for an optocoupler, specifically an MID-400 [26]. A reference LED will turn on or off in the presence of an AC voltage. The LED causes a phototransistor on another circuit to turn on or off. In this case, it allows the APC to convert the presence of an AC voltage to a DC voltage. A current limiting resistor is placed at the AC input of the optocoupler. The DC output of the optocoupler contains a pull-up resistor. The pull-up resistor provides a high-level voltage when the output of the transistor is in the off condition. A capacitor is used to keep the gate of the transistor from oscillating due to the AC line frequency.

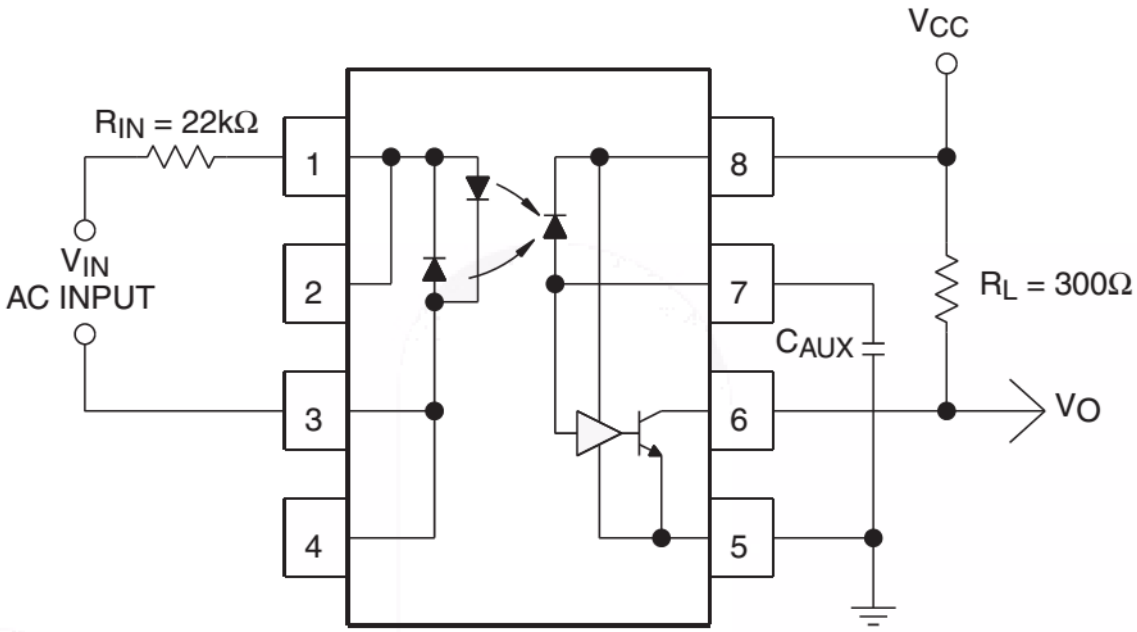

Figure 2.2: Example optocoupler circuit–MID-400 [26].

With this implementation, the 120VAC output to each pedestrian signal must be connected to an AC detector on the APC. Once the processor in the APC has determined the state of the pedestrian signals by monitoring the DC outputs of the optocouplers, it communicates the state to each APB. One method to relay this information is to use Ethernet Over Power-Line (EOP) technology. This method of communication allows the APC to transmit to the APBs using existing pedestrian button cables [27], [28]. Figure 2.3 shows a block diagram of the APC. Figure 2.4 shows a diagram of AAPS implementations where N specifies the number of signal lights in the intersection.

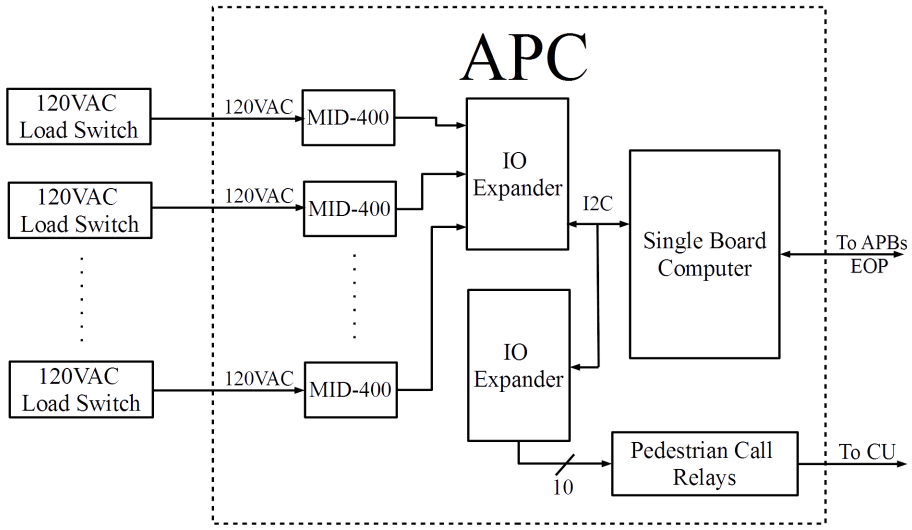

Figure 2.3: APC block diagram.

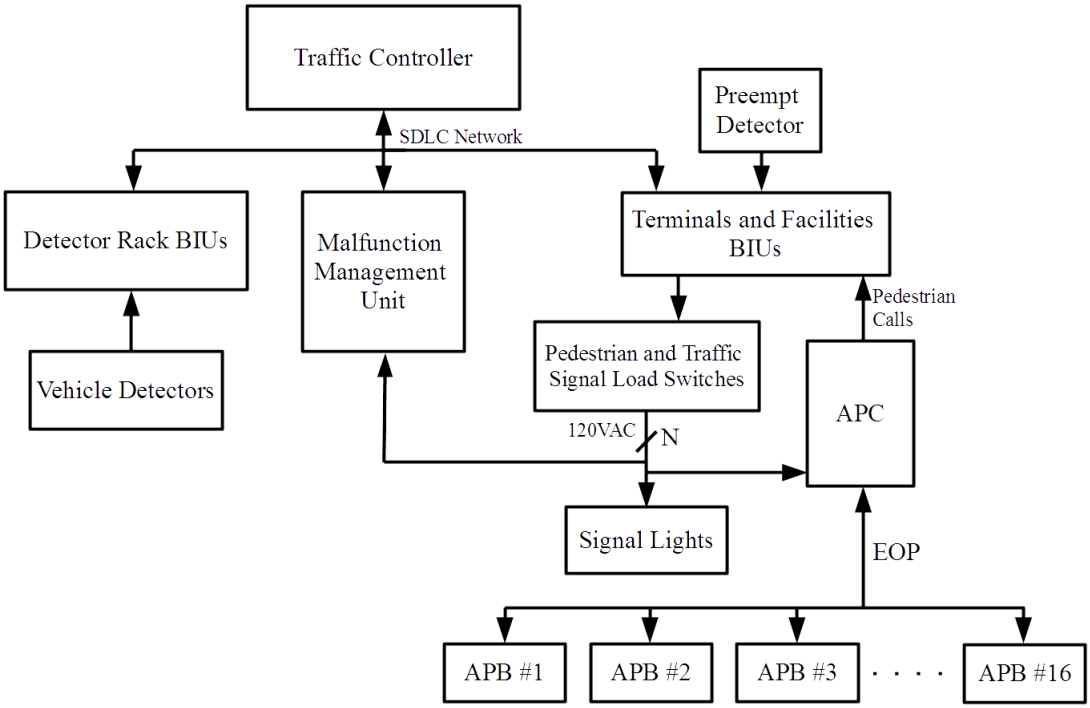

Figure 2.4: Example of a current APS implementation.

The present implementation of the AAPS results in inconsistent information among different equipment within a traffic cabinet for a short time (see Chapter 6 for examples). The information inconsistency is partially due to different voltage thresholds among devices to detect when a traffic signal is on or off. The MMU is another device that

actively monitors 120VAC load switch outputs for signal lights. The MMU and AAPS use different voltage thresholds to determine when a signal is on or off. Table 2.2 shows the different voltage thresholds. Furthermore, the circuitry the AAPS uses to sense the state of pedestrian signals will put a small electrical load on each load switch output the AAPS is connected to. This small load could potentially cause the MMU to detect a false ON.

Device	**ON Voltage (RMS)**	**OFF Voltage (RMS)**
MMU (Green, Yellow, W)	>25	<15
MMU (Red, DW)	>60 ± 10	<15
AAPS (MID-400)	100	30 (Estimate)

Table 2.2: On and off AC RMS voltage thresholds among devices in a traffic cabinet.

Another potential area for error in the current implementation of AAPS is the complicated wiring involved in installing the system. Since each pedestrian light requires its own individual wire routed to the APC within the traffic cabinet, the abundance of wires can cause confusion. The technician may accidentally connect to the wrong load switch or fail to connect a signal light altogether. Figure 2.5 shows just how complicated the inside of a traffic cabinet can be.

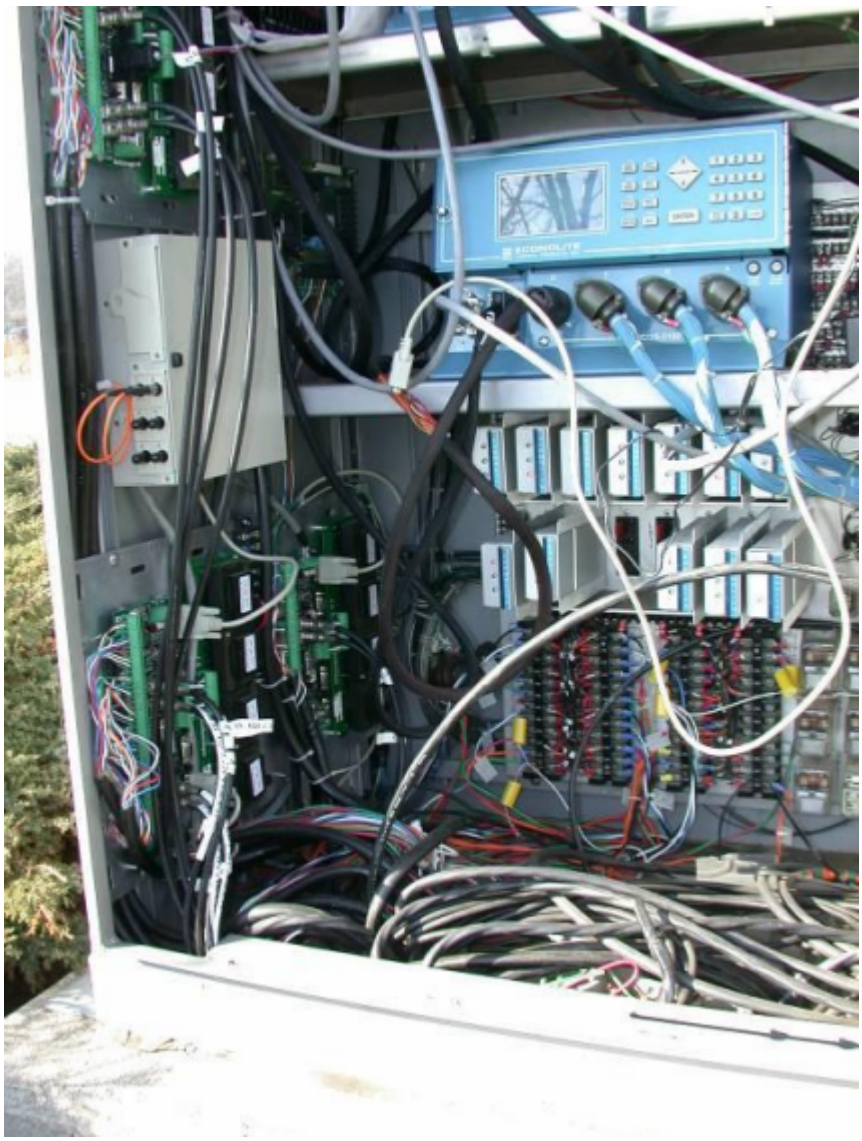

Figure 2.5: The inside of a typical traffic cabinet [29].

2.2.2 Proposed Next Generation AAPS

One of the benefits of the proposed next generation AAPS is the elimination of the complicated direct connections to 120VAC load switches which results in less hardware. Eliminating the parallel load caused by the AAPS system reduces the chance the MMU may incorrectly detect a conflict.

Improved information consistency is attributed to the method the new AAPS uses to decode pedestrian signal states. The new AAPS implementation detects pedestrian

signal status via the SDLC network within NEMA TS2 traffic controller cabinets. Specifically, the Type 129 Response message from the MMU to the traffic controller on the SDLC network is exploited by the AAPS to obtain pedestrian signal information.

The Type 129 Response message contains all signal information as interpreted by the MMU. The CU requests the Type 129 message every 100 milliseconds. Utilizing the Type 129 message, the AAPS system avoids inconsistency in signal decoding that could arise due to the different voltage thresholds between devices discussed earlier. The AAPS will receive the signal information from the MMU at the same time as the CU.

Other packets could have been utilized such as messages transmitted from the CU to the BIUs controlling signal lights. However, these messages only specify what the CU wants the signals to be. They do not reflect the actual state of the signals. There is some delay in the transmission of the CU's message to when the signals actually change. In a pedestrian system, it is imperative to obtain the state of the intersection as it physically is, not what it is soon going to be. The Type 129 message from the MMU meets this requirement.

The threshold voltages shown in Table 2.2 are from an AAPS that uses an MID-400, an AC line monitor logic-out device [26], to sense whether a particular pedestrian signal is on or off. Due to a large current limiting resistance in the AAPS (62kΩ), the threshold for an ON voltage is higher than listed in the MID-400 manual. This is because the MID-400 manual assumes a smaller current limiting resistance in series to the input of the MID-400 (22kΩ). The MID-400 manual does not explicitly give an OFF voltage threshold. Figure 2.6 is used to estimate that the OFF voltage threshold is about 30VAC RMS with a 62kΩ resistor [26]. It is shown later in the results section of this thesis that the MID-400 implements some delay before it shows that a signal has turned off.

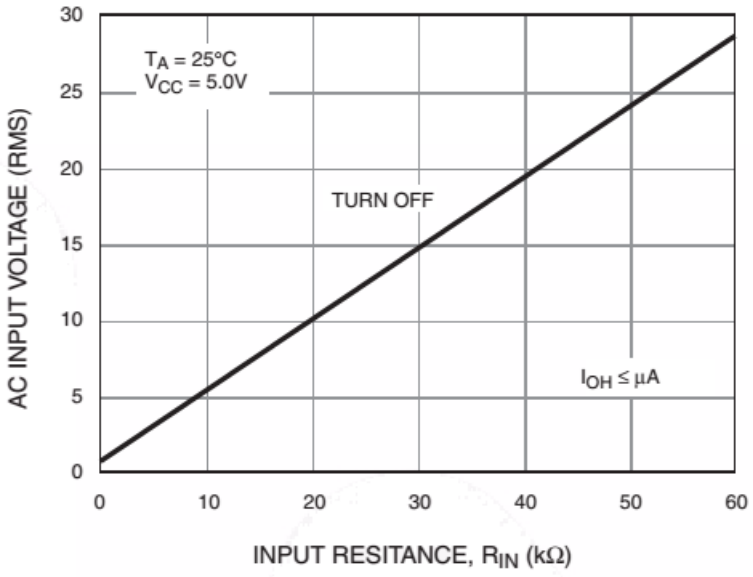

Figure 2.6: Threshold voltages for MID-400 [26].

It is to be noted that the MMU has different voltage thresholds for red and Don't Walk (DW) signals than it does for green, yellow, and Walk (W) signals. This is a desirable function of the MMU. Because a green, yellow, or W traffic signal will indicate to the driver or pedestrian that he or she has the current right of way, the MMU needs to be particularly sensitive to these signals.

Figure 2.7 shows a block diagram of the new AAPS. The new APC consists of three devices: a single-board computer running a Linux operating system, an EOP communications method, and the new AAPS SDLC interface hardware. The single-board computer has numerous responsibilities such as hosting a web page for easy maintenance and configuration of the AAPS as well as controlling communications between the APC and the APBs via EOP. The AAPS SDLC interface hardware allows the APC to sense pedestrian signal information from the Type 129 Response message from the MMU.

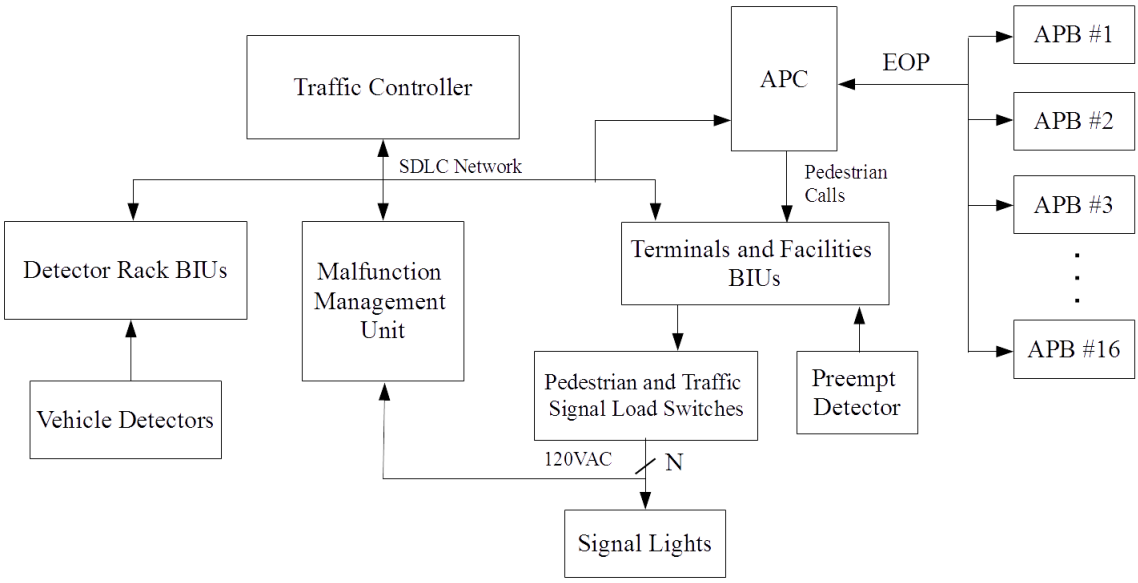

Figure 2.7: Block diagram of new AAPS.

The new AAPS implementation utilizes a CPLD to detect when a Type 129 Response message from the MMU is transmitted to the traffic controller. Once the Type 129 message is detected, all the pedestrian signal information is decoded by the CPLD and placed into data registers within the CPLD's architecture. These data registers can be accessed via the Inter-integrated Circuit (I2C) serial communications protocol by the single-board computer within the APC.

Chapter 3: NEMA TS2 SDLC Network

The June 1986 definition of SDLC by IBM is as follows: "SDLC is a discipline for managing synchronous, code-transparent, serial-by-bit information transfer between nodes that are joined by data links" [22]. The SDLC protocol supports both full-duplex and half-duplex communications. Full-duplex communications allow for data to be communicated in both directions simultaneously. Half-duplex only allows communication in one direction at a time. The SDLC network used in NEMA TS2 traffic cabinets is capable of full duplex communications.

The SDLC network within NEMA TS2 traffic cabinets consists of one primary device and multiple secondary devices in a master/slave configuration. The primary (master) device is typically a CU. Secondary (slave) devices are the MMU and BIUs. BIUs are used to interface devices such as traffic lights and vehicle detectors to the SDLC bus network. Figure 3.1 shows a block diagram of the devices connected to an SDLC bus network within traffic cabinets. Although TF BIUs 5 through 8 and DR BIUs 5 through 8 are listed, they are defined as "Reserved," "Spare," or "Manufacturers Use" in the NEMA TS2 specification [17].

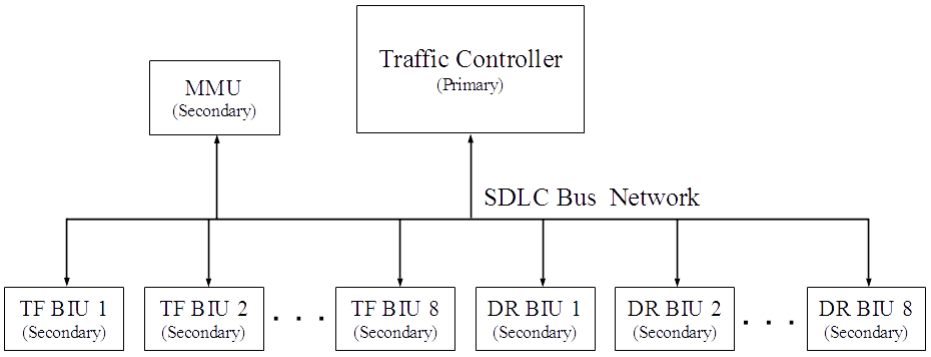

Figure 3.1: NEMA TS2 SDLC bus configuration.

3.1 SDLC: Physical Layer Description

The physical part of the SDLC communications network within NEMA TS2 traffic cabinets conforms to the EIA-485 standard (also known as RS-485 or TIA-485). EIA-485 is a multi-point bus network that allows for multiple drivers and receivers to be connected to the same physical pairs of wires [34]. Physically, EIA-485 is a differential signal interface. In NEMA TS2 traffic cabinets there are two differential pairs of wires for communications out of the traffic controller and two differential pairs of wires for communications from secondary devices. Each differential pair consists of a clock and data signal. Although this allows full-duplex communications, secondary devices typically start to transmit only once they have received a complete message from the traffic controller. The EIA-485 standard grants a maximum of 32 unit loads to be connected to the bus [34]. See Figure 3.2 for a wiring diagram of the SDLC network where N can be up to 32 devices. Each pair of wires is a differential pair.

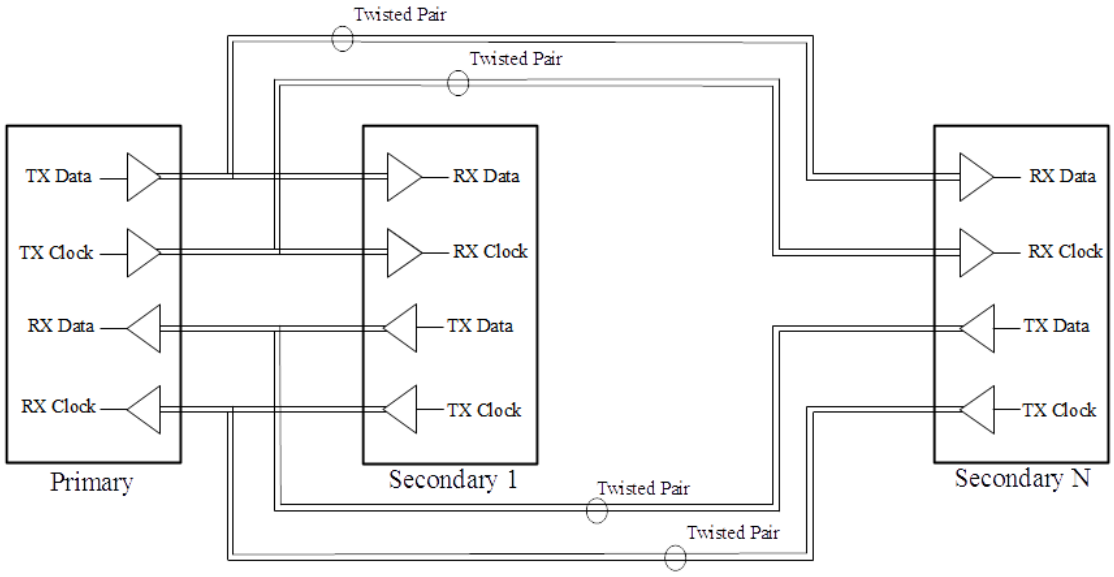

Figure 3.2: NEMA TS2 SDLC network wiring diagram.

3.2 SDLC: Message Format

All information required to obtain the current state of a signalized intersection is communicated on the SDLC network. The information transferred between the traffic controller and secondary devices includes detector information, traffic signal status, pedestrian call status, and preempts.

The frame format of an SDLC message is shown in Figure 3.3. The beginning and end of a frame is represented by the hexadecimal value 0x7E. After the opening flag, the 8-bit address of the secondary device is sent. Addresses of all secondary devices are shown in Table 3.1. When the CU transmits a message, it puts the address of the secondary device to receive the message in the address field. When a secondary device responds to the CU, the device puts its address in the address field. Next, a control byte is sent. The control byte is always set to the hexadecimal value 0x83. Variable length information is sent next. There is no limit on the length of the information field, but it must be a multiple of eight bits in length. The information field is at least one byte long to specify the message type. A 16-bit cyclic redundancy check (CRC) is sent after the variable length information. Finally, the message ends with a closing flag of 0x7E.

0x7E	8-bits	0x83	1 to N-bytes	16-bits	0x7E
Opening Flag	Address	Control	Variable Length Information	CRC	Closing Flag

Figure 3.3: NEMA TS2 SDLC message frame.

Device	Address	Device	Address	Device	Address
TF BIU 1	0	TF BIU 7	6	DR BIU 5	12
TF BIU 2	1	TF BIU 8	7	DR BIU 6	13
TF BIU 3	2	DR BIU 1	8	DR BIU 7	14
TF BIU 4	3	DR BIU 2	9	DR BIU 8	15
TF BIU 5	4	DR BIU 3	10	MMU	16
TF BIU 6	5	DR BIU 4	11	Diagnostic	17

Table 3.1: Device addresses on NEMA TS2 SDLC bus.

The SDLC protocol utilizes zero bit insertion after a consecutive five binary 1s are transmitted. Zero bit insertion takes place in all message fields except for the opening and closing flags. If a receiving device detects either a zero bit insertion error or a CRC error, the message will be ignored.

3.2.1 Cyclic Redundancy Check

William Stallings defines CRC as follows: "Given a k-bit block of bits, or message, the transmitter generates an (n-k)-bit sequence, known as a Frame Check Sequence (FCS), such that the resulting frame, consisting of n bits, is exactly divisible by some predetermined number. The receiver then divides the incoming frame by that number and, if there is no remainder, assumes there was no error" [35]. The CRC operation can be performed using a shift register with an XOR operation between bits specified by an FCS polynomial. The binary polynomial for computing the FCS in NEMA TS2 SDLC packets is $X^{16} + X^{12} + X^5 + 1$. The FCS is computed for the entire packet except for the opening and closing flags. The FCS shift register is shown in Figure 3.4. The shift register starts with all values (C0 to C16) initialized to a binary 1.

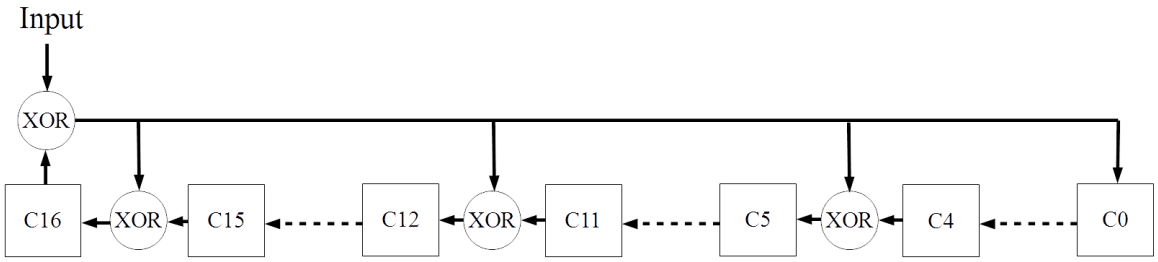

Figure 3.4: SDLC packet FCS computation.

On a receiving device, each bit received, excluding zero insertion bits and opening and closing flags, is shifted through the FCS shift register. The receiving device also computes its FCS field on the FCS value the transmitting device has sent. If no errors occur, the resulting value of the FCS field (bits C0 to C15) at the receiver should always be calculated as the hexadecimal value 0xF0B8. The FCS the receiver computes is always the same regardless of what data has been sent if no error occurred. This is because the receiver computes the FCS on the entire message frame including the transmitter's computed FCS value. As the receiver is computing the FCS, the value should be calculated as the same value the transmitter calculated. However, the receiver needs to check that the FCS the transmitter sent is free of errors. Therefore, the receiver continues to compute its FCS as the transmitter is sending the CRC of its packet. Because of this, the receiver should always compute 0xF0B8 as its FCS value if no errors occur. If the resulting value does not equal 0xF0B8, the receiver ignores the contents of the message because an error occurred. The FCS computation is explained in detail in Appendix B of [22].

Using a CRC field in a message packet is acceptable for information integrity, but it is not sufficient for data security. A clever hacker would be able to change specific bits in the message for the FCS to still be calculated as 0xF0B8 while still corrupting the message. Ming Yu et al. show an approach for a more secure SDLC message passing architecture [36].

3.3 Message Types

There are many types of messages transmitted on the SDLC bus of NEMA TS2 CAs. Table 3.2 and Table 3.3 summarize the messages passed on the SDLC bus as described in the NEMA TS2 standard [17]. Command frames are generated by the CU and are sent to the appropriate secondary device. Response frames are generated by secondary devices and are sent to the CU. Secondary devices do not respond to any broadcast message by the CU. Secondary devices cannot directly communicate to other secondary devices, but they can send information to each other via the CU through message types 42, 43, 169, 170, and 171.

Type	**Destination**	**Function**
0	MMU	Load Switch Drivers
1	MMU	MMU Inputs/Status Request
3	MMU	MMU Programming Request
9	ALL	Data And Time Broadcast
10	TF BIU #1	Outputs/Inputs Request
11	TF BIU #2	Outputs/Inputs Request
12	TF BIU #3	Outputs/Inputs Request
13	TF BIU #4	Outputs/Inputs Request
18	ALL	Output Transfer Frame Broadcast to TF BIUs
20	DR BIU #1	Call Data Request
21	DR BIU #2	Call Data Request
22	DR BIU #3	Call Data Request
23	DR BIU #4	Call Data Request
24	DR BIU #1	Reset/Diagnostic Request
25	DR BIU #2	Reset/Diagnostic Request
26	DR BIU #3	Reset/Diagnostic Request
27	DR BIU #4	Reset/Diagnostic Request
42	Any Secondary	Secondary Destination Message
43	Any Secondary	Secondary Exchange Status

Table 3.2: NEMA TS2 Command Frames [17].

Type	**Source**	**Function**
128	MMU	Type 0 ACK
129	MMU	Inputs/Status
131	MMU	Programming
138	TF BIU #1	Inputs
139	TF BIU #2	Inputs
140	TF BIU #3	Inputs
141	TF BIU #4	Inputs
148	DR BIU #1	Call Data
149	DR BIU #2	Call Data
150	DR BIU #3	Call Data
151	DR BIU #4	Call Data
152	DR BIU #1	Diagnostic
153	DR BIU #2	Diagnostic
154	DR BIU #3	Diagnostic
155	DR BIU #4	Diagnostic
169	Any Secondary	Secondary Source Message
170	Any Secondary	Secondary NACK
171	Any Secondary	Secondary ACK

Table 3.3: NEMA TS2 Response Frames [17].

Chapter 4: Hardware Overview

Obtaining the state of a signalized intersection can be done via the SDLC network of NEMA TS2 traffic cabinets using certain hardware. For the applications discussed in this thesis, a CPLD was chosen to interface a Single Board Computer (SBC) to the SDLC network. The CPLD monitors the SDLC network and stores all desired information within internal data registers that the SBC can access via the Inter-integrated Circuit (I2C) serial communications protocol. The SBC is capable of hosting a web page that allows the user to configure the device and monitor activity on the SDLC network. See Figure 4.1 for a diagram of the components involved in interfacing to the SDLC network.

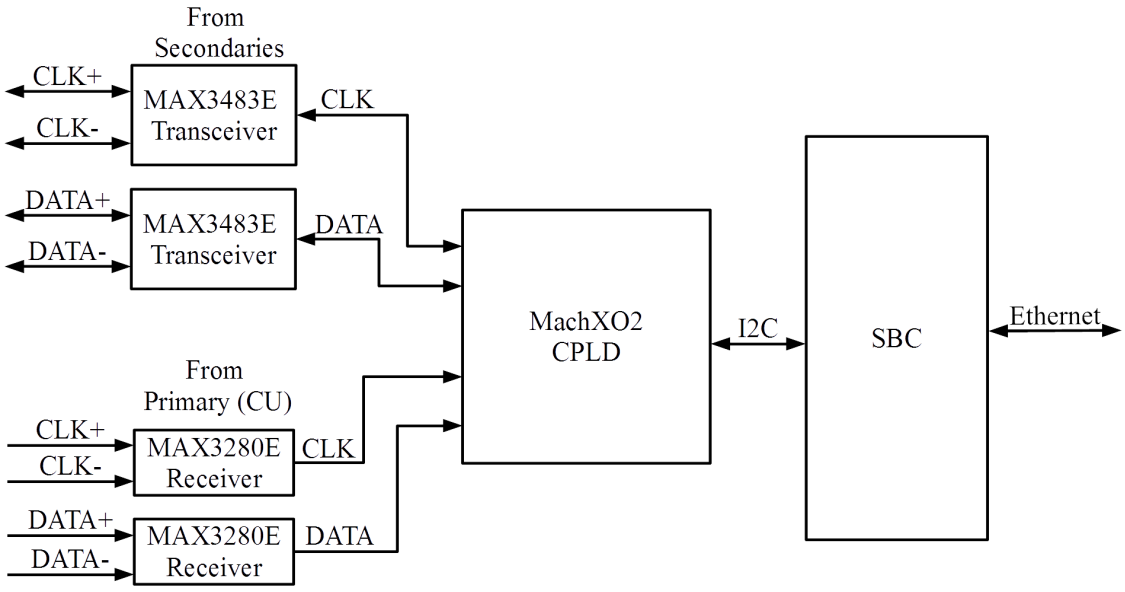

Figure 4.1: SDLC interface block diagram.

4.1 EIA-485 Translation

The EIA-485 standard utilizes differential signal pairs for each individual signal. Since the SDLC network conforms to the EIA-485 standard at the physical level, the differential signals of the network are converted to single ended signals before being connected to the CPLD. To convert the clock and data signals of secondary devices on the SDLC network, a MAX3483E transceiver manufactured by Maxim Integrated is

used [37]. A transceiver allows the transmission of messages as well as the reception of messages. Transmission of messages is a requirement of this architecture if it is ever implemented as an active secondary device on an SDLC network such as a CID or an APS system with functionality not currently defined in present day APS systems.

A receiver is used to convert differential clock and data signals on the SDLC network of the primary device to single ended signals. A receiver rather than a transceiver is used because the applications do not require the emulation of primary message transmissions. Both CID and APS applications discussed in this thesis assume a CU is present, eliminating the need for either the CID or APS to transmit messages as a primary device. The specific receiver chosen for these applications is the MAX3280E receiver manufactured by Maxim Integrated [38].

4.2 Complex Programmable Logic Device

The CPLD chosen to interface to the SDLC network is the Lattice MachXO2 manufactured by Lattice Semiconductor [39]. The MachXO2 contains Programmable Function units (PFUs) for custom hardware designs and an Embedded Function Block (EFB). The EFB includes a hardware timer, serial communication blocks such as I2C, and a Wishbone Interconnect that gives the PFUs an interface to the EFB. The specifics of the Wishbone Interconnect standard are described by OpenCores [40]. For the applications described in this thesis, it is used to access I2C, Serial Peripheral Interface (SPI), and Timer control registers. See Figure 4.2 for a block diagram of the MachXO2 architecture.

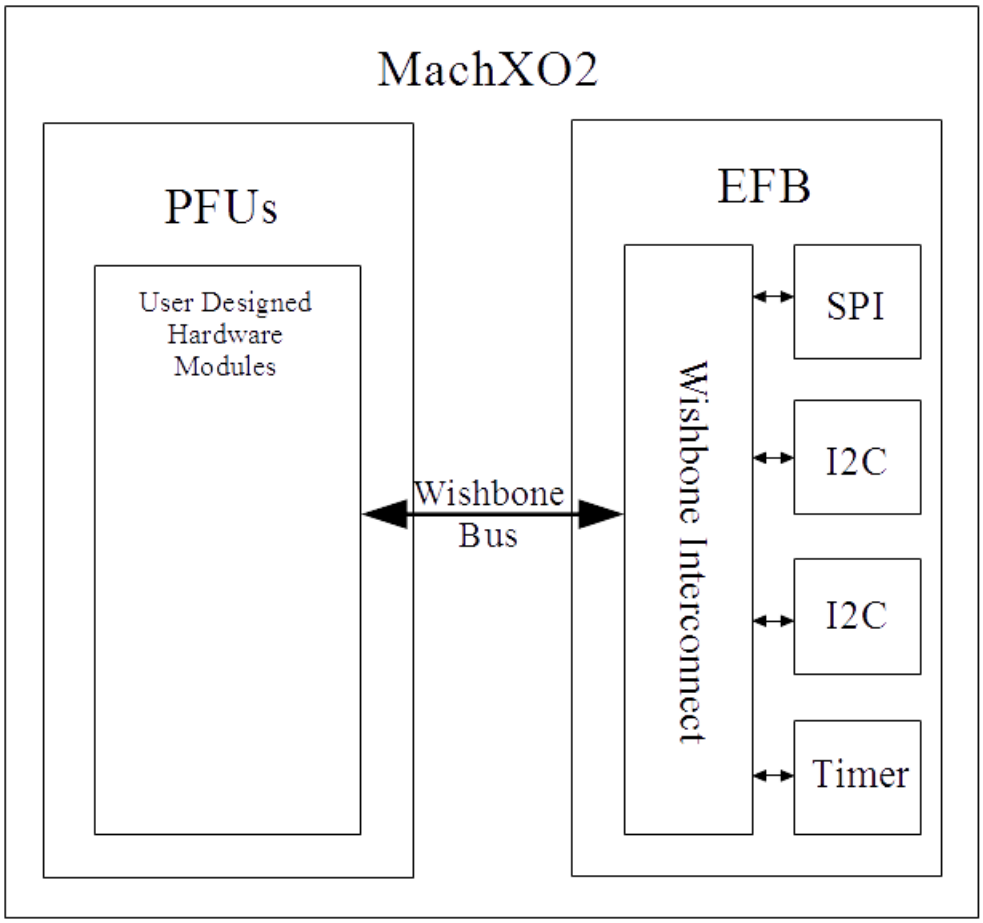

Figure 4.2: MachXO2 CPLD Architecture.

The MachXO2 contains a Digitally Controlled Oscillator (DCO) that is used to generate its internal system clock. It can be configured for a variety of frequencies ranging from 2.08 MHz to 133.00 MHz [39]. For the applications described in this thesis, the CPLD has been configured for 10.23 MHz. The frequency of the SDLC network is at 153.6 KHz. This gives the CPLD about 66 clock cycles per bit received on the SDLC network to receive and store each bit of an SDLC message frame. Using the DCO allows the CPLD to operate with no external crystal or oscillator required. The downside to using the DCO is that it is less accurate than a crystal oscillator. It is important to note that increasing the DCO's frequency requires more power than lower frequencies. Using more power is not a concern as the device will be connected to mains power through the traffic cabinet or wall outlet.

The EFB was a major reason for choosing the MachXO2 in this study. Without the EFB, serial communication interfaces would have had to be designed in an HDL. The EFB allows the use of serial communications without the need to design a serial communications controller. Furthermore, the EFB enables the programming of the MachXO2 via its serial communication interfaces. The MachXO2 has an internal non-volatile program flash that is accessible via many different programming methods through the EFB including I2C [41]. This is useful if part of the design is changed and the user desires to update the CPLD while it is in use. The SBC can retrieve the new program file for the CPLD and upgrade it via I2C, for example.

The NEMA TS2 standard implements strict timing requirements for communications on the SDLC network. The frequency of communications must be 153.6 KHz $\pm$ 1% [17]. A CPLD was chosen over a microcontroller because the timing requirements on the SDLC network favored a more parallel design approach. For example, it is possible that a microcontroller could be interrupted during an SDLC transaction and violate the SDLC timing requirements. The PFUs within the CPLD could be configured in such a way that the SDLC network could be accessed at the same time an I2C transaction takes place without interfering with each other. This makes the CPLD more deterministic than a microcontroller for this design.

An Application Specific Integrated Circuit (ASIC) could have been designed to do the work of the CPLD. However, it would have been expensive and time-consuming to design and manufacture. The time and cost of designing an ASIC would need to be justified for these applications. For example, it could be reasonable to justify the extra resources required to design an ASIC if many of these devices were produced and sold. For the goals of this research project, however, the MachXO2 CPLD will suffice.

4.2.1 Wishbone Interconnect

The Wishbone interconnect within the CPLD's EFB is used to integrate internal data registers with the I2C serial communications interface for the applications discussed in this thesis. This allows an external device to access information from the CPLD contained in its internal data registers. Two Finite Sate Machines (FSM) have been designed to control the transfer of data from the user created registers in the PFU to the I2C registers in the EFB. The FSMs are designed in the the HDL Very High Speed Integrated Circuit Hardware Description Language (VHDL).

The first FSM, the Wishbone master, controls what registers are undergoing a data transfer. The Wishbone master FSM is shown in Figure 4.3. The FSM input is a source register address and a destination register address. These register addresses can be either a user created register within the PFUs or a register within the EFB. For the applications discussed in this thesis, there are two types of registers for the Wishbone interface contained within the PFUs: output data registers and input data registers. Wishbone output data registers are capable of read and write operations from the Wishbone interconnect, whereas the Wishbone input data registers are only capable of being read from the Wishbone interconnect. The VHDL for the output registers is shown in Listing B.6. The VHDL for the input registers is shown in Listing B.7.

A temporary Wishbone output data register, r0, is used to read data out of the I2C status register in the CPLD's EFB. This register contains the information shown in Table 4.1. The bits important for this application are the BUSY bit, SRW bit, and the TRRDY bit. The BUSY bit is set to a binary 1 if the I2C module within the CPLD is undergoing a transaction. The SRW bit indicates if a read or write transaction is taking place. A binary 0 specifies a write operation, and a binary 1 specifies a read operation. The TRRDY bit indicates if the transmitter or receiver data register in the I2C interface on the CPLD is ready for more data. A binary 1 specifies that the I2C transmitter or receiver is ready. For more information, see the MachXO2 Family Handbook [39].

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Name	TIP	BUSY	RARC	SRW	ARBL	TRRDY	TROE	HGC

Table 4.1: MachXO2 I2C status register bits [39].

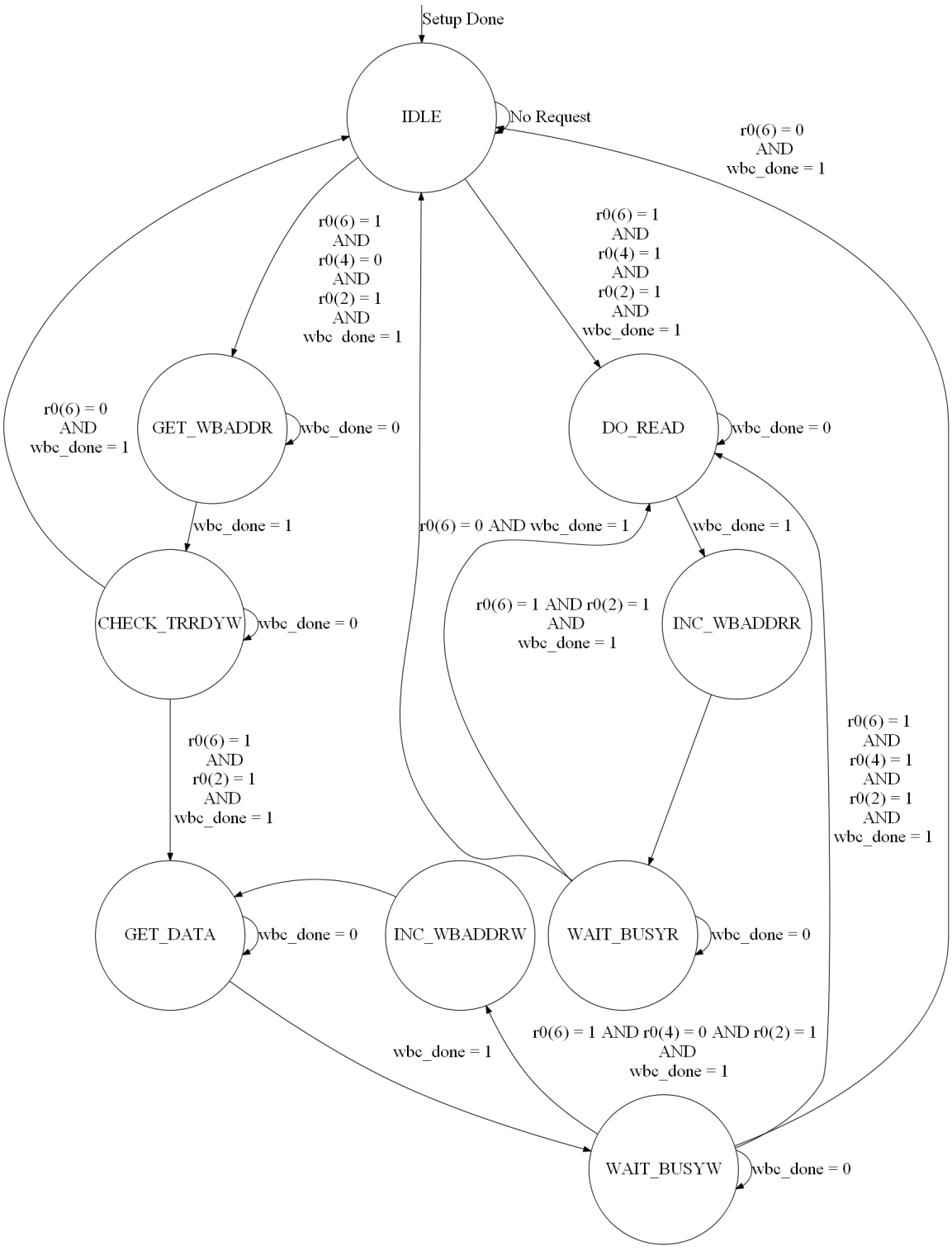

Figure 4.3: Wishbone master finite state machine.

Upon system reset, the Wishbone master FSM transitions the CPLD through a series of states that setup the functionality of the I2C communications interface in the EFB.

Once setup is complete, the Wishbone master FSM waits in the IDLE state until a transfer is initiated by the I2C interface in the EFB. In order to check if a transfer is requested or not, the Wishbone master FSM constantly monitors the I2C status register within the EFB. In the I2C protocol, the first byte transmitted contains the 7 bit device address as well as a read/write bit. If the read/write bit is a binary 0, a write operation is requested. If the read/write bit is a binary 1, a read operation is requested. The contents of the I2C status register are continually transferred to the temporary Wishbone register r0.

The IDLE state transitions to the DO_READ state if a master I2C device has requested to read a register in the CPLD. A read operation is known if bit 4 of r0 is set to a binary 1. It is also checked that the I2C interface on the CPLD is ready to receive or transmit data via the TRRDY bit in bit 2 of register r0. The DO_READ state transfers data from the requested register from an I2C master device to the I2C transmit buffer in the CPLD.

Once an individual byte has been read from the CPLD by an I2C master device, the Wishbone master FSM transitions to the INC_WBADDRR state. This state will automatically increment the internal register address by one if the I2C master device requests to read subsequent values.

After the register address has been incremented, the Wishbone master FSM changes to the WAIT_BUSYR state. This state waits for the Wishbone interconnect to transfer data from internal data registers to I2C registers. Once the transfer is complete, the WAIT_BUSYR state checks if more data is requested by the I2C master device. If so, the Wishbone master FSM transitions back to the DO_READ state. If not, it transitions to the IDLE state and waits for another request from an I2C master device.

If in the IDLE state and an I2C master device requests to write data to the CPLD, the Wishbone master FSM transitions to the GET_WBADDR register to read the register address. In I2C communications, the second byte received just after the device address

is the internal register address of the device.

Once the internal register address is read from the Wishbone interconnect, the Wishbone master changes to the CHECK_TRRDYW state. This state checks if the master I2C device wants to write data to the CPLD. This is done by checking the I2C BUSY bit in the status register via bit 6 of register r0. If the bit is a binary 1, the I2C master is continuing to write data to the CPLD. This causes the Wishbone master FSM to transition to the GET_DATA state. Otherwise, the Wishbone master FSM returns to the IDLE state.

The GET_DATA state reads one byte that the I2C master device writes to the CPLD. Once one byte is read, the FSM transitions to the WAIT_BUSYW state.

The WAIT_BUSYW state checks if the I2C master device has more data to write to the CPLD. If so, the FSM transitions to the INC_WBADDRW state to increment the internal register address to the next register to be written to in the CPLD. If not, the FSM returns to the IDLE state. The VHDL for the Wishbone master FSM is shown in Listing B.4.

The second FSM, the Wishbone controller, is used to directly control the signals on the Wishbone interconnect to enable the transfer of data between the PFU and the EFB. The state diagram for this FSM is shown in Figure 4.4.

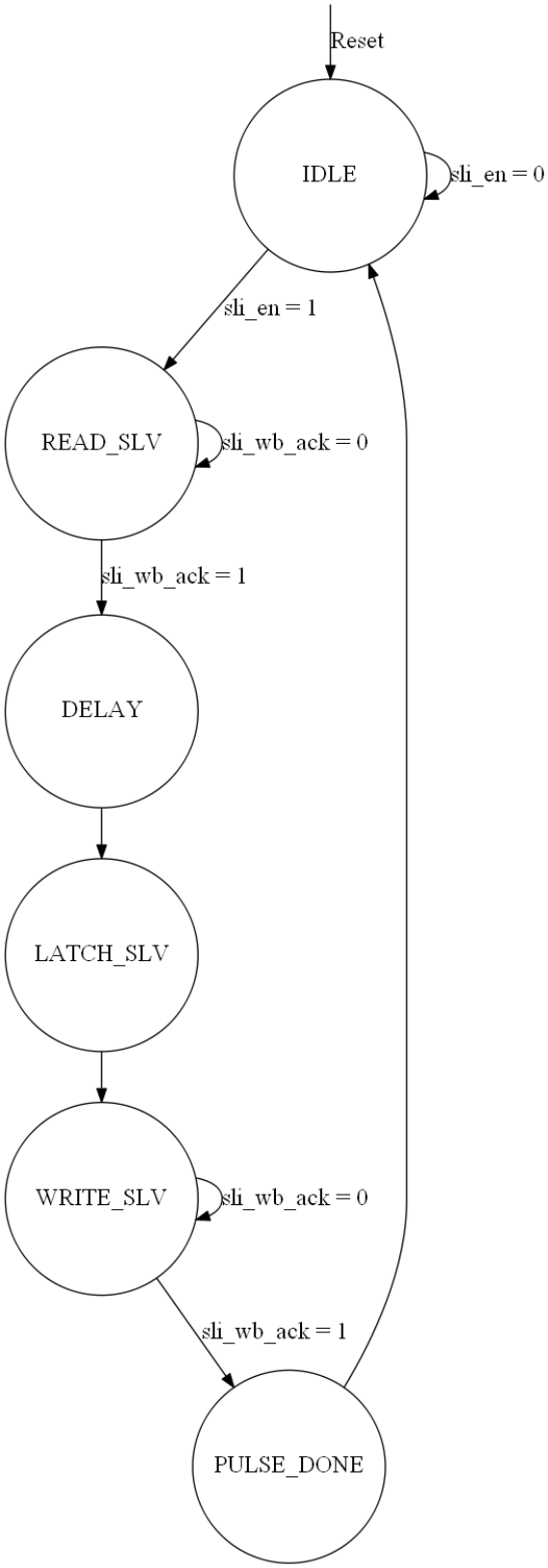

Figure 4.4: Wishbone controller finite state machine.

Once the Wishbone master FSM is configured with source and destination register addresses, it enables the second FSM, the Wishbone controller, to control Wishbone interconnect signals. The Wishbone master enables the Wishbone controller by asserting the Wishbone controller's sli_en flag.

Upon system reset, the Wishbone controller is initialized to the IDLE state. This state waits for sli_en to be asserted before transitioning to the READ_SLV state.

The READ_SLV state asserts the Wishbone signals slo_wb_cyc and slo_wb_stb to indicate to the Wishbone interconnect that a read operation is requested. Asserting both slo_wb_cyc and slo_wb_stb signals is required to initiate a data transfer even though the signals appear to have the same operation. This is because the Wishbone standard has multiple modes of operation described in [40]. The READ_SLV state also sets the address on the Wishbone interconnect to the source register address specified by the Wishbone master FSM. Once the EFB acknowledges the request by asserting the sli_wb_ack signal, the Wishbone controller changes to the DELAY state. This state gives the Wishbone interconnect enough time to complete the transfer of data between the PFU and EFB.

The DELAY state unconditionally transitions to the LATCH_SLV state. The LATCH_SLV state deasserts the slo_wb_cyc and slo_wb_stb signals. This causes the data to be latched into a temporary register within the CPLD's PFU.

Once the data is latched, the Wishbone controller changes to the WRITE_SLV state where the slo_wb_cyc, slo_wb_stb, and slo_wb_we signals are asserted. The slo_wb_we signal specifies a write operation on the Wishbone interconnect. The WRITE_SLV state also sets the address on the Wishbone interconnect to the destination register address specified by the Wishbone master FSM.

Once the Wishbone interconnect acknowledges that the transfer of data is complete from the WRITE_SLV state, the Wishbone controller transitions to the PULSE_DONE state to assert the slo_done signal. The slo_done signal is used to indicate to the Wishbone master FSM that a data transfer on the Wishbone interconnect has been completed.

The VHDL for the Wishbone controller FSM is shown in Listing B.5.

4.3 SDLC Packet Receiver

A NEMA TS2 SDLC packet receiver is designed in VHDL to run on the CPLD. It monitors the SDLC network for packets of interest. An FSM is used to control the reception of an SDLC frame described in Chapter 3. The FSM enables different hardware modules within the CPLD that parse SDLC message frames for desired information. The top level signal diagram of an SDLC packet receiver, named sdlcdecoder, for use in an AAPS is shown in Figure 4.5. The sdlcdecoder module is shown in Listing B.2 in Appendix B. The module checks for a Type 129 response message from the MMU and outputs the corresponding red and green signal statuses for each traffic signal including pedestrian signals. All components and signals instantiated within the sdlcdecoder are shown in Figures 4.6, 4.7, and 4.8. All VHDL for SDLC packet reception is shown in Appendix B for an AAPS.

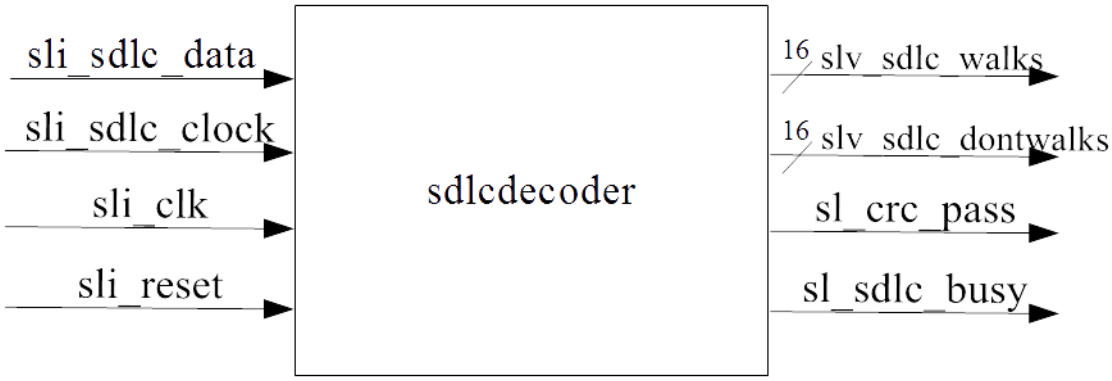

Figure 4.5: SDLC receive signal diagram–top level signals.

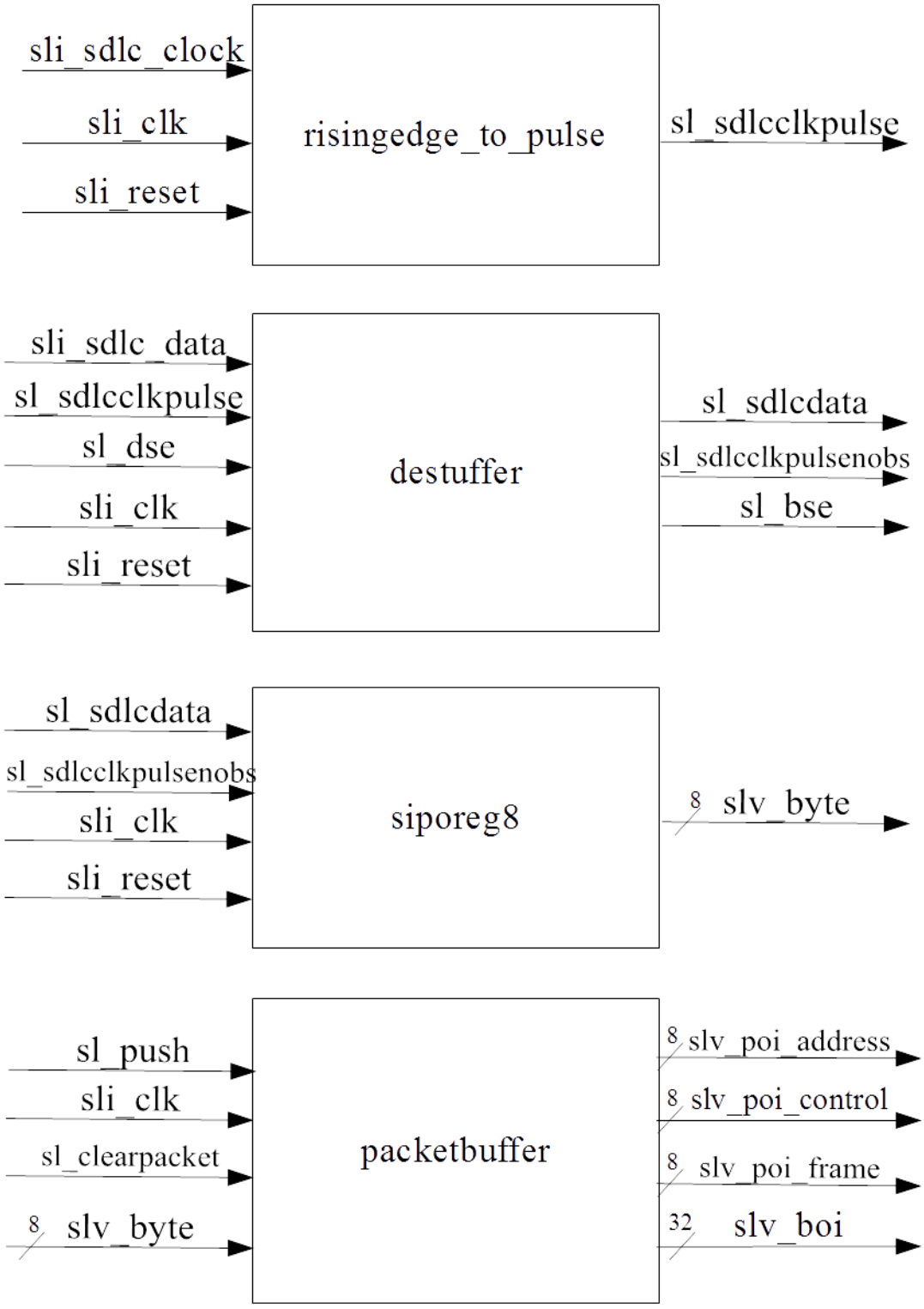

Figure 4.6: SDLC receive signal diagram 1.

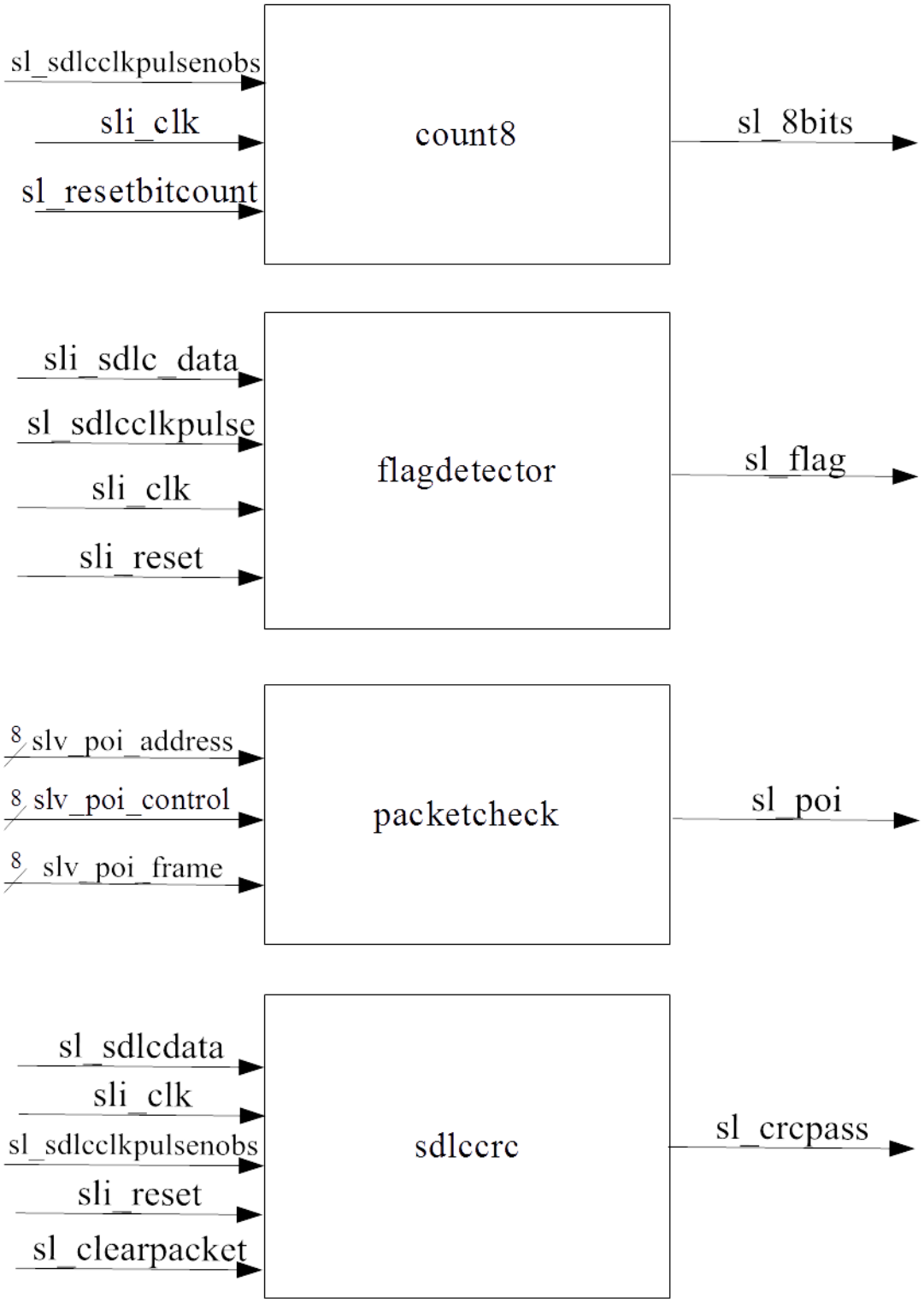

Figure 4.7: SDLC receive signal diagram 2.

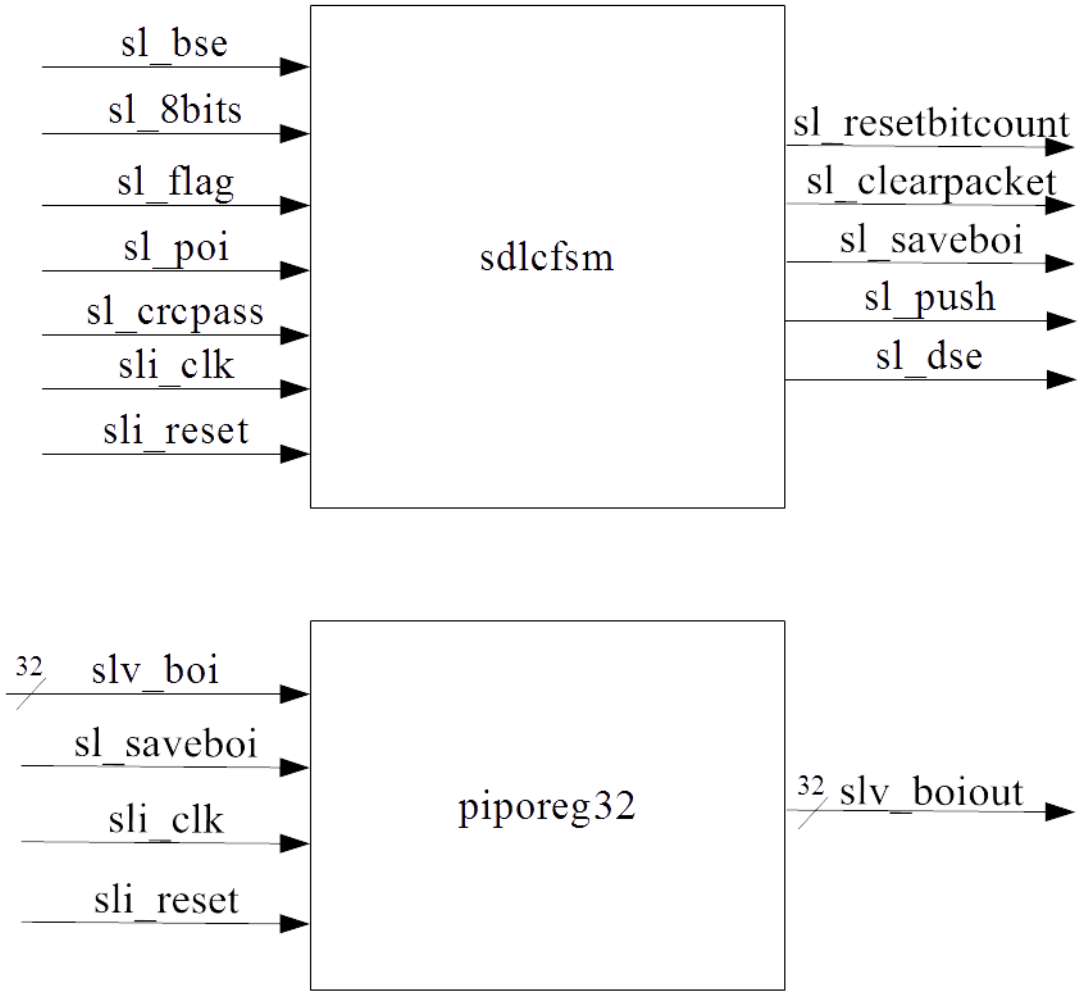

Figure 4.8: SDLC receive signal diagram 3.

The risingedge_to_pulse module converts the rising edge of the SDLC clock signal to a single pulse the same width of the CPLD's system clock which runs at 10.23 MHz. This is used to trigger various shift registers throughout the CPLD to shift on the next bit of information from the SDLC network. The VHDL for this module is shown in Listing B.10 in Appendix B.

The destuffer module removes zero bit insertion bits. If a zero bit insertion bit is detected, the module keeps the slo_pulse signal from the risingedge_to_pulse module from being asserted to the other modules. In this manner, the destuffer module freezes the receive modules until the next SDLC bit is received if a zero bit insertion bit is detected. This keeps the error detection bit from being buffered in the information registers. The

VHDL for the destuffer module is shown in Listing B.13 in Appendix B.

The siporeg8 module is a simple 8-bit serial in, parallel out shift register. It simply buffers 8-bits at a time as they are received from the SDLC network. The VHDL for the siporeg8 module is shown in Listing B.11 in Appendix B.

The packetcheck module is used to check if the device address, control byte, and SDLC frame type are all set to the desired values for a specific SDLC message frame of interest. In the case of an AAPS, the packet of interest has an address of 16, control byte of 0x83, and an SDLC frame type of 129 to specify the Type 129 response message from the MMU. If all of these values are received, the packetcheck module asserts the slo_poi signal to specify that the current message being received is a packet of interest. The VHDL for the packetcheck module is shown in Listing B.17 in Appendix B.

The flagdetector module checks for 0x7E values which are the values of an opening or a closing flag. The VHDL for the flagdetector module is shown in Listing B.14 in Appendix B.

The sdlccrc module is used to check for a valid CRC flag when it is received from the SDLC message. If a CRC error is detected, the CPLD ignores the current SDLC message being received. The VHDL for the sdlccrc module is shown in Listing B.12 in Appendix B.

The count8 module keeps track of how many bits have been received from the SDLC network (not including zero bit insertion bits). Once eight bits have been received, the module asserts the slo_count8 signal. The VHDL for the count8 module is showin in Listing B.9 in Appendix B.

The sdlcfsm module is the finite state machine that runs the overall reception of an SDLC message frame. It is explained in detail in the next section.

The piporeg32 module is a 32 bit parallel in, parallel out data register that buffers information bits of interest. In this case, it buffers all red and green traffic signal bits from the MMU's Type 129 response message. This is used by an AAPS to determine

the state of pedestrian signals. The VHDL for the piporeg32 module is shown in Listing B.18 in Appendix B.

4.3.1 SDLC Packet Receive State Machine

An FSM has been designed in VHDL to control SDLC frame reception. An entire SDLC frame is buffered within internal 8-bit data registers before being transferred to Wishbone registers accessible via I2C. To do this, the FSM is inputted the following flags from other modules: a bit stuff error flag (sl_bse), an eight bits received flag (sl_8bits), a flag to specify that an SDLC opening flag has been detected (sl_flag), a packet of interest flag (sl_poi), and a CRC pass flag (sl_crcpass). The sl_bse flag is set when a zero bit insertion error occurs. The sl_8bits flag is set when eight bits are received and gets reset when those eight bits are buffered in an internal data register. The sl_poi flag is set when the CPLD has successfully decoded a packet type from the information field of the SDLC frame that the device is interested in storing in Wishbone registers accessible via I2C. Depending on the application, some packets do not need to be stored by the CPLD. Unnecessary packets do not set the packet of interest flag.

The state diagram of the SDLC packet receive FSM is shown in Figure 4.9. The FSM includes five states: IDLE, IN_PACKET, SAVE_BYTE, SAVE_BOI, and CHECKEOP. The VHDL for the FSM for use in an AAPS application is shown in Listing B.19 in Appendix B.

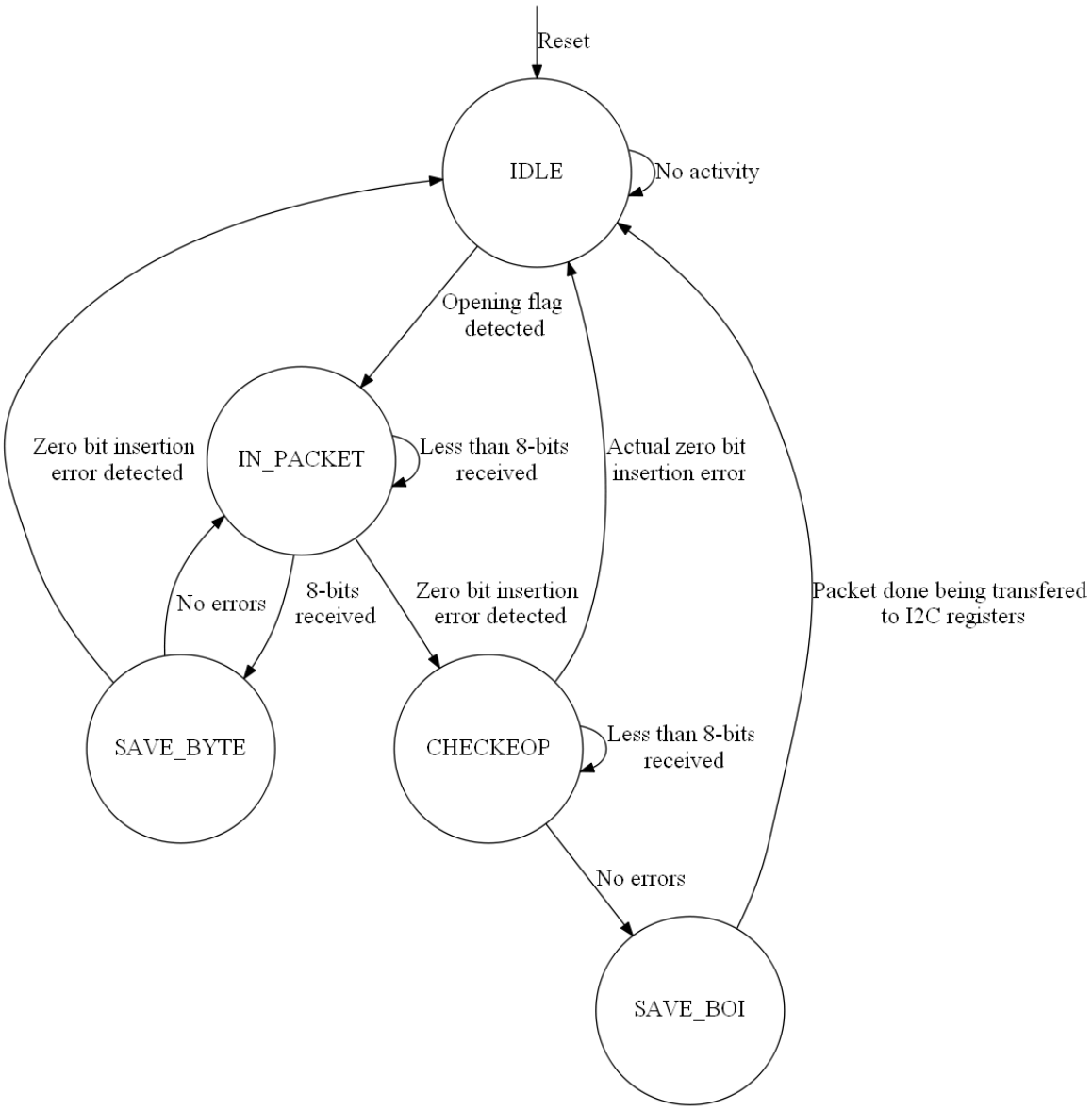

Figure 4.9: SDLC receive finite state machine.

The IDLE state resets internal data registers and waits for an opening flag (0x7E) to be received. Once an opening flag is detected, the FSM transitions to the IN_PACKET state which specifies to other modules to start buffering received bits. Each byte is sequentially stored in 8-bit data registers until the entire frame has been received. An example packet buffer for use in an AAPS is shown in Listing B.21 in Appendix B. For an AAPS, the packet buffer contains 15 8-bit parallel in, parallel out data registers to

buffer the entire SDLC message frame. The number of registers required depends on what message frame the CPLD is storing. The largest message frame on the NEMA TS2 SDLC network is the Type 148-151 message from DR BIUs [17]. This message contains 37 bytes in the information field. Therefore, the maximum amount of 8-bit parallel in, parallel out data registers the CPLD would require is 37. Data registers rather than flash memory are used due to the frequency of accessing these registers. The data in these registers also needs be accessed quickly to transfer the data to Wishbone registers accessible via I2C. Receiving 8-bits triggers the FSM to transition to the SAVE_BYTE state. The SAVE_BYTE state enables the packet buffer module to sequentially buffer the byte.

As bits are being received, the IN_BYTE state checks for zero bit insertion errors. If a zero bit insertion error is detected, the FSM ignores subsequent bits and waits for the next opening flag. If the FSM has received an entire frame and a zero bit insertion error occurs, it transitions into the CHECKEOP state to check if an end of packet flag was the last byte received. If the value 0x7E was the last byte received, it is assumed that the zero bit insertion error was caused by the closing flag and no actual error occurred since an entire packet has been buffered.

Opening and closing flags will cause the zero bit insertion error flag to be asserted because they contain six subsequent binary 1s. Therefore, the zero bit insertion error flag is used to check for a closing flag. A closing flag specifies the end of the SDLC frame, and the FSM transitions into the SAVE_BOI state to save the information field of the packet. The SAVE_BOI state enables the transfer of the buffered bytes of interest into Wishbone registers. The registers designed in VHDL and accessible via I2C are shown in Listing B.7 and Listing B.6.

Received bytes are buffered rather than immediately stored in Wisbhone registers in case an error occurs during reception of the frame. This allows the CPLD to completely receive a packet and check for CRC and zero bit insertion errors before storing anything

in Wishbone registers. This reduces the chance that invalid data is retrieved from the CPLD by the SBC.

While in the SAVE_BOI state, the signal output by the FSM to transfer the bytes of interest into Wishbone registers is driven by a CRC check module. If the CRC bytes of the frame are valid, then the information field bytes are stored. Conversely, if the CRC bytes of the frame are invalid, the information field bytes are ignored.

There is a chance that an SDLC frame is being stored by the CPLD as an I2C transaction is taking place. This can cause data read from the CPLD via I2C to be conflicting as it may change during the data transfer. However, the NEMA TS2 standard states that conflicts of no longer than 450 milliseconds must be tolerated [17]. Therefore, the conflict that may occur due to SDLC and I2C transactions taking place at the same time are not of a concern.

4.3.2 Zero Bit Insertion Removal

Raw data being received by the CPLD needs to be checked for zero bit insertions. Between opening and closing flags of an SDLC frame, a binary 0 is inserted after five subsequent binary 1s. Zero bit insertion ensures that an opening and closing flag of an SDLC frame cannot appear anywhere else within the SDLC message [22]. This is because the opening and closing flags are specified by the binary value 01111110 which contains more than five consecutive binary 1s. If more than five consecutive binary 1s are received other than in the opening and closing flags, a zero bit insertion error occurred and the entire frame is ignored.

The "destuffer" module is a 6-bit serial shift register. All raw data within an SDLC frame is shifted through the destuffer register. If a binary 0 has been shifted onto the register and is followed by five consecutive binary 1s, the binary 0 will be ignored by holding the clock pulse of the packet buffer high. This ensures that the binary 0 is not stored in the buffer as it was just an error detection bit and not an information bit.

4.3.3 Cyclic Redundancy Check Hardware

In order to check for CRC errors, a CRC decoder was designed in VHDL and is shown in Listing B.12 in Appendix B. The CRC decoder is a 17-bit serial in parallel out shift register. Upon system reset, the entire register is initialized to binary 1s. Once an opening flag of an SDLC frame is received, each bit received thereafter is shifted through the CRC check register up to the closing flag. Once the entire frame has been shifted through the CRC check register, the remaining value is checked. If the value does not equal the hexadecimal value 0xF0B8 as explained in Subsection 3.2.1, the entire frame will be ignored. When the SDLC network is idle, the CRC check register gets reset to all binary 1s.

4.4 SDLC Packet Transmitter

If an application requires SDLC packet transmission such as a CID, a device that emulates all secondary devices on the SDLC network, or an AAPS with functionality not currently defined, then the CPLD must be able to transmit messages on the SDLC network. Therefore, the Printed Circuit Board (PCB) for use in an AAPS shown in Appendix A allows the transmission of SDLC messages by using EIA-485 transceivers as discussed previously.

There is some risk to including transceivers when an application does not require SDLC message transmission. If the device fails in an active state, it may unintentionally be transmitting bits out of the transceivers, causing the entire SDLC network to become corrupted. The direction of the EIA-485 transceivers is controlled by two pins on the MAX3483E: a read enable pin and a driver enable pin. The different configuration outcomes of these pins is shown in Table 4.2. A binary 1 represents a high logic level (+3.3 volts) and a binary 0 represents a low logic level (zero volts). An X represents a "don't care" bit, as in it doesn't matter whether it is a binary 0 or a binary 1.

Read Enable Pin	**Driver Enable Pin**	**Direction State**
0	0	Receive Enabled
X	1	Transmit Enabled
1	0	High Impedance

Table 4.2: MAX3483E Transceiver Direction Pins [37].

As an application that requires SDLC message transmission has not been investigated in as much detail as one with SDLC message reception, it is not described in detail in this thesis. However, all the required hardware necessary at the physical layer for SDLC message transmission has been described.

Chapter 5: Advanced Pedestrian System Implementation

Many messages transmitted on the SDLC network include the state of signal lights of the intersection. For an AAPS, the Type 129 response message from the MMU is used over other message frames. This message was chosen because it contains the physical state that a pedestrian signal is currently in as monitored by the MMU [17].

The desired data from the information field of a Type 129 message is bits nine through 24 and bits 41 through 56, as described in the NEMA TS2 standard [17]. Bits nine through 24 contain the green signal information for every channel. Bits 41 through 56 contain the red signal information for every channel. In terms of a pedestrian signal, the green signal information corresponds to a Walk signal, and the red signal information corresponds to a Don't Walk signal. Though not all bits stored by the CPLD correspond to pedestrian signals, there is no current standard to specify what channels are to be used for pedestrian signals as opposed to vehicle signals. Furthermore, no yellow signal information from bits 25 to 40 from the Type 129 message are used. At some signalized intersections, the yellow signal information bits specify a Flashing Don't Walk (FDW) signal for pedestrian signals, but this is not a standard and therefore is not used in this application. Hence, it is the job of the SBC's software to determine which of the channel bits are implemented as pedestrian signals.

5.1 Hardware Flashing Don't Walk Decoding

A FDW pedestrian signal may be implemented differently between different signalized intersections. For example, some intersections assert a solid yellow signal when the DW signal is flashing for a given load switch even though only the red and green signals are used for a pedestrian signal. However, this is not common at every intersection. In order to accurately decode FDW signals for an AAPS, the CPLD senses if the DW signal from the Type 129 message is physically flashing. Performing this logic assures that an FDW signal is accurately detected within any signalized intersection that uses

an SDLC network containing the Type 129 message.

Logic has been designed within the CPLD in VHDL to determine the state of an FDW signal from the W and DW signals being received from the Type 129 message. If a W signal is active, the FDW signal state is known to be off and no further logic needs to be performed. If a W signal is not active, then the DW signal is sampled every 100 milliseconds to determine if it is flashing. A 10-bit serial-in, parallel-out shift register is used to determine if the DW signal is flashing. Each DW signal state that is sampled over each 100 millisecond interval is shifted onto the shift register. A binary 1 represents the DW signal being on and a binary 0 represents the DW signal being off. If some of the values in the shift register are a binary 0 and some are a binary 1, then the FDW signal state for the appropriate channel is set to a binary 1. In other words, this indicates that the DW signal is currently flashing. If all of the bits in the shift register are either a binary zero or a binary one, then the DW signal is not flashing, and the FDW signal state for the appropriate channel is set to a binary 0.

The 10-bit shift registers are loaded with the W and DW signal information extracted from the Type 129 message. The same W and DW data that gets loaded into Wishbone registers is loaded into the 10-bit shift registers for FDW decoding. Listing B.20 shows the individual 10-bit shift registers implemented in VHDL. Listing B.3 shows the VHDL used to load each individual shift register from the W and DW signal information. The FDW decoder in B.3 outputs the FDW signal information in a 16 bit vector which is then loaded into Wishbone registers along with the W and DW signal information.

Figure 5.1 shows an example of different shift register states used to determine the state of an FDW pedestrian signal. Figure 5.1a shows the output of the FDW decoder when the signal is in the W state. Figure 5.1b shows the output of the FDW decoder when the signal is in the FDW state. Figure 5.1c shows another FDW example. Figure 5.1d shows the output of the FDW decoder when the signal is in the DW state.

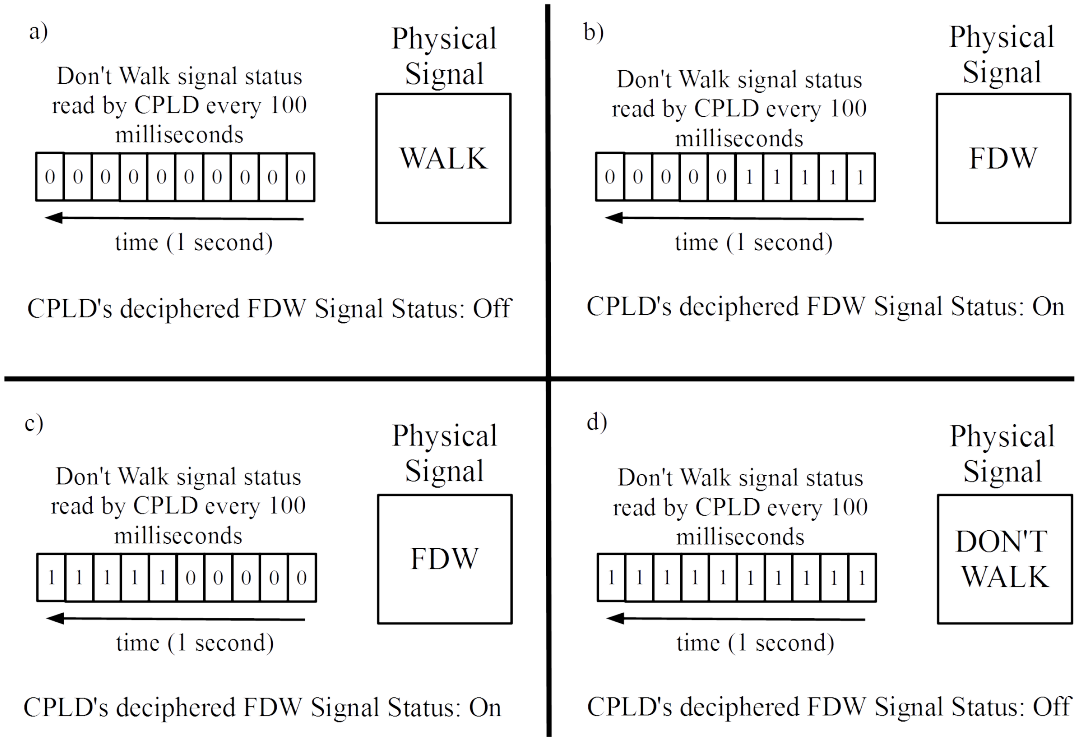

Figure 5.1: Flashing don't walk signal decoding.

In order to achieve the 100 millisecond sample period, the EFB's timer in the MachXO2 CPLD is used. This timer can be reconfigured during runtime via the Wishbone Interconnect, which would be useful at a signalized intersection that may not conform to MUTCD 2009 standards.

As described in Section 4E.04 of MUTCD 2009, an FDW signal may flash at a rate of no less than 50 times per minute or no more than 60 times per minute [2]. This means that the DW signal is on for about half a second and off for about half a second when flashing. Since the 10-bit shift register contains DW signal information for one second, it can accurately be used to determine the FDW state. It is to be noted that this implementation of FDW decoding can cause a maximum one second delay between a sensed FDW signal by the SDLC interface board and the physical FDW signal. This means that the AAPS may still be sensing an FDW signal for at most one second while the physical signal has transitioned to the DW state.

Chapter 6: Testing and Results

A comparison of traffic signal decoding methods was done. The MID-400 method used by present day AAPSs is compared to the method utilizing the MMU via the SDLC network in NEMA TS2 traffic cabinets. Testing was performed using an Econolite ASC/3-2100 NEMA TS2 Type 2 traffic controller and an MMU-16LE(ip) MMU by Eberle Design, Inc. Figure 6.1 shows a block diagram of the test setup.

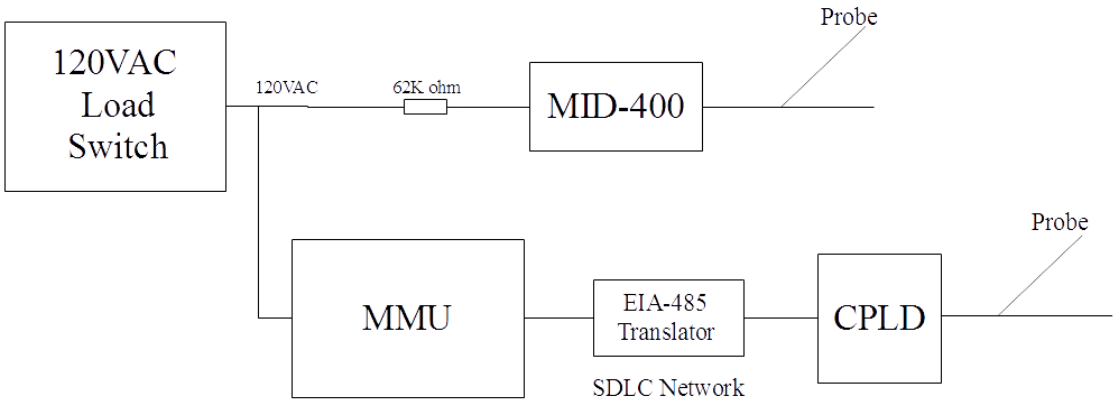

Figure 6.1: Test setup for comparison of signal decoding times.

6.1 Comparison of Signal Decoding Time

Since Type 129 messages are only polled every 100 milliseconds by the traffic controller, the worst case scenario in delay for detecting a change occurs when the signal changes just after the Type 129 message is sent by the MMU. This will add up to 100 milliseconds to the delay it takes the MMU to sense that a signal has changed. Figure 6.2 and Figure 6.3 show this added delay occurring when compared to the output of an MID-400. A high voltage level indicates that the signal was sensed as off, where a low voltage level indicates that the signal is on. Note that these tests do not account for the time it takes for an APB to receive and decode a message regarding signal state from the APC.

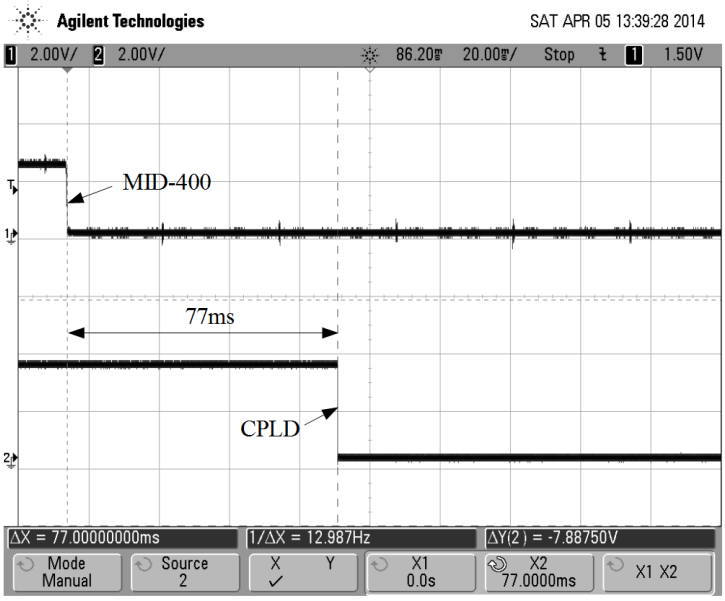

Figure 6.2: MMU signal sensing best case delay.

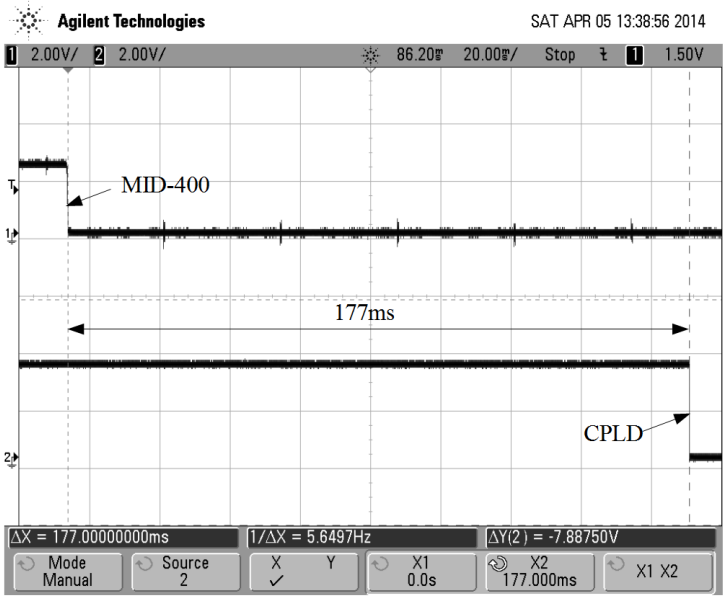

Figure 6.3: MMU signal sensing worst case delay.

The MID-400 is designed in such a way that there is a slight delay from when an AC voltage is no longer sensed to when it changes output. The delay is a result of the filter to remove the pulses at zero crossings of the AC signal. Otherwise, the zero crossing of the AC signal would cause the digital output of the MID-400 to oscillate. This delay can cause the MMU to detect that a traffic signal has turned off before the MID-400 does. Even with the added delay of the CPLD deciphering the Type 129 message, it can occasionally detect the signal change faster than the MID-400. Figure 6.4 shows an example of this occurring. This is a good example of showing how, for a short time, information can be inconsistent among different devices in the traffic cabinet. The average human response time to a change in visual information is about 445 milliseconds [44], which shows that either method of signal detection is performing faster than a human could, regardless of the method.

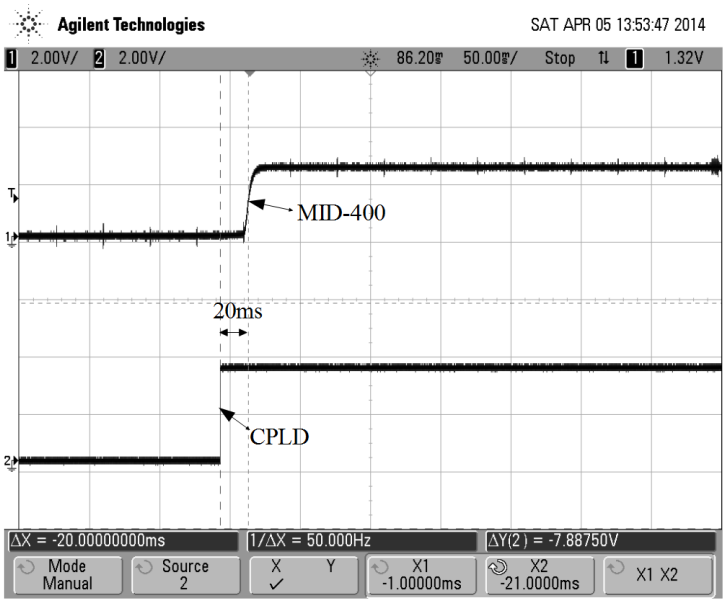

Figure 6.4: MMU sensing faster than MID-400.

6.2 Flashing Don't Walk Detection Testing

To test the FDW detection module, the DW, W, and FDW states were outputted by an SBC as they were retrieved via I2C every 100 milliseconds. As shown in Figure 6.5, as the DW signal is oscillating (flashing), the FDW signal is high, indicating that the CPLD is decoding the FDW state correctly. In the figure, if the signal is high it is on. If the signal is low it is off.

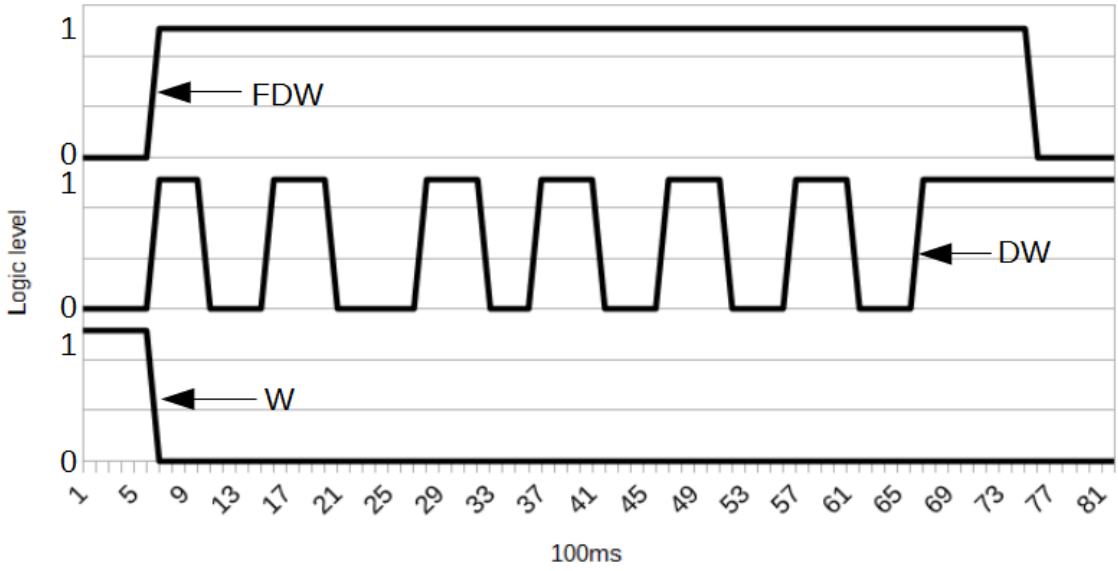

Figure 6.5: FDW Decoding.

6.3 System Reliability Comparison

Equation 6.1 describes reliability in terms of the number of components required to have a fully functioning system where R_s is the total reliability of the system, R_i is the reliability of an individual component, and N is the total number of components required for the entire system to work [42]. Equation 6.1 shows that it is detrimental to the system's reliability to add components as the new component's reliability gets multiplied into the system's total reliability.

$$R_s = \prod_{i=1}^{N} R_i \tag{6.1}$$

In terms of the hardware necessary for the proposed SDLC method of real-time traffic signal instrumentation, fewer components are required than present day instrumentations. Considering an AAPS system that instruments four crossings, it would require four MID-400s to sense the W signal lights and four MID-400s to detect the DW signal lights. The SDLC method of instrumentation only requires two transceivers and the CPLD (the receivers are not necessary in this case as messages from the CU will not need to be used). This is true for any size of intersection being instrumented. The older method required two MID-400s for each traffic signal being instrumented at the intersection. Therefore, the larger the intersection the less reliable the system becomes with the MID-400 method of instrumentation.

The individual component reliability can be estimated from the number of pins on each component. This method of calculating reliability is only an estimate. For a more rigorous calculation of reliability, other factors should be considered such as number of transistors and transistor size, among others. The Military Handbook on Reliability Prediction of Electronic Equipment, or MIL-HDBK-217F describes the failure rate of microcircuits with a given number of functional pins [43]. This failure rate is used to find the R_i value for Equation 6.1. The reliability of a component is simply $1 - F$ where F is the component's failure rate. The equation described in the MIL-HDBK-217F to calculate the failure rate is shown in Equation 6.2 where N_p is the number of pins on the device [43].

$$F = 2.8 \times 10^{-4}(N_p)^{1.08} \tag{6.2}$$

The components included in the reliability calculation for the new SDLC method of traffic signal instrumentation are shown in Table 6.1. The estimated failure rate for

components in the old AAPS method is shown in Table 6.2. Table 6.2 includes an I2C expander. An SBC accesses the state of each MID-400 via an I2C expander. An I2C expander is used to serialize the output of parallel data signals into a serial information string via I2C. There are two IO expanders used: one for W signals and one for DW signals.

Component	**Number of Pins**	**Failure Rate**
CPLD	144	0.060
Transceiver	8	0.003

Table 6.1: Failure rate of components for new method.

Component	**Number of Pins**	**Failure Rate**
MID-400	8	0.003
I2C Expander	24	0.009

Table 6.2: Failure rate of components for old method.

Given the values from Table 6.1 and using Equation 6.1, the total reliability for the new SDLC traffic signal instrumentation is shown in Equation 6.3.

$$\begin{aligned} R_{new} &= (1 - 0.060) \times (1 - 0.003)^2 \\ R_{new} &= 0.934 \end{aligned} \tag{6.3}$$

Using the values from Table 6.2, the total system reliability for pedestrian traffic signal instrumentation using MID-400s is described in Equation 6.4. N is the number of traffic signals to be instrumented, showing that reliability will decrease as more pedestrian signals are instrumented.

$$R_{old} = (1 - 0.003)^{2N} \times (1 - 0.009)^2$$
$$R_{old} = 0.892 \text{ for } N = 16 \tag{6.4}$$

Table 6.3 summarizes the reliability comparison of the old versus new method of traffic signal instrumentation for an AAPS. As shown by the table, the old method is only more reliable in small intersection implementations. The SDLC traffic signal instrumentation method's reliability stays the same regardless of the size of the instrumentation. This is because the information for each signal light is gathered from the SDLC network, not from a physical connection to the signal light as in the MID-400 method.

Method	**Reliability**
SDLC	0.934
MID-400: 4 Signals	0.959
MID-400: 8 Signals	0.936
MID-400: 12 Signals	0.914
MID-400: 16 Signals	0.892

Table 6.3: Reliability Comparisons.

Note that the reliability calculations shown in Table 6.3 do not include other components of the two systems being compared. It is assumed that the rest of the system will be approximately the same. Furthermore, the above calculations do not include the number of connections between components. If a more strict reliability comparison is desired, more testing and analysis needs to be done.

The Military Handbook MIL-HDBK-217F is a handbook published in 1992. The information gathered from it may be out of date. However, for the purposes of this work the information from MIL-HDBK-217F is adequate for a rough comparison of two different systems.

Chapter 7: Conclusions and Future Work

7.1 Conclusion

It has been shown that the SDLC network within NEMA TS2 traffic cabinets can be utilized for real time traffic signal instrumentation. This method keeps information among devices within a traffic cabinet more consistent among each other as opposed to the MID-400 method used by present day AAPSs. The information is kept consistent by utilizing the decoding method of the MMU rather than using an independent method.

Even though the two different approaches discussed result in different delays of signal sensing shown in Chapter 6, the differences in delay may not be large enough to greatly influence human reaction time. However, the proposed method for a second generation AAPS can be more reliable as it has fewer components. Furthermore, the SDLC instrumentation method is easier to install making it less likely that a mistake will occur during the installation of the system.

In addition, the SDLC method of instrumentation is well within the 32 unit load maximum for an EIA-485 signal interface as only nine secondary devices (four TF BIUs, four DR BIUs, and the MMU) are explicitly defined in the NEMA TS2 standard. The older method interfaced to 120VAC load switches, which adds an electrical load to the load switch output. The extra load from the instrumentation circuitry may cause the MMU to miss malfunctions in the signal lights, or detect malfunctions that didn't actually happen.

Failure of the SDLC traffic signal instrumentation should be considered. If the SDLC method fails in an active or noisy state, it will corrupt the entire SDLC network. If that happens, it will trigger the traffic controller to put the intersection into a flash mode of operation. This is undesirable, and more testing should be done to determine the best way of avoiding this type of failure.

7.2 Future Work

The SDLC method of traffic signal instrumentation discussed in this thesis has been designed and tested for an AAPS. However, only the SDLC interface board has been designed at the point of this writing. Work has been started on software for an SBC to interface to the SDLC interface board. A pedestrian system has been tested using the SDLC method of signal instrumentation. However, as stated previously, more software on the SBC needs to be designed to allow configuration of what traffic signals are used for pedestrian signals. The system used to test the SDLC method of signal instrumentation had the pedestrian signal mapping hard-coded in the software. However, more work is to be done to complete an entire AAPS that effectively uses the SDLC method of instrumentation.

Work has been started on an SDLC interface that emulates all secondary devices within a NEMA TS2 traffic cabinet for a CID application. However, there is still work to do in that regard. Present day CIDs are used to interface to the large A, B, and C connectors in older style traffic cabinets, however they are bulky and out of date devices. Researchers desire to have a smaller CID that can interface to NEMA TS2 controllers through the SDLC network.

The same device that is being developed as a CID can be placed in an actual traffic cabinet and log information retrieved from the SDLC network. The single board computer can log information gathered from the SDLC network within a database. This is desirable so researchers can monitor the functionality of a signalized intersection without the need to have expensive camera systems installed or hiring a person to monitor traffic flow by hand.

References

[1] A. Santos *et al.*, "Summary of Travel Trends: 2009 National Household Travel Survey," Federal Highway Administration, Washington D.C., Rep. FHWA-PL-11-022, Jun. 2011.

[2] *Manual on Uniform Traffic Control Devices for Streets and Highways*, 2009 ed. 2nd rev., Federal Highway Administration, Washington D.C., 2012.

[3] B.E. Chander *et al.*, "Signalized Intersections: An Informational Guide," Federal Highway Administration, Washington D.C., Rep. FHWA-SA-13-027, Jul. 2013.

[4] R. L. Gordon *et al.*, "Traffic Control Systems Handbook," Dunn Eng. Assoc., Westhampton Beach, NY, Rep. FHWA-HOP-06-006, 2005.

[5] L. Zhao and C.E. Thorpe, "Stereo- and Neural Network-Based Pedestrian Detection," *IEEE Transactions on Intelligent Transportation Systems*, vol. 1, no. 3, pp. 148-154, Sep. 2000.

[6] F. Suard *et al.*, "Pedestrian Detection using Infrared images and Histograms of Oriented Gradients," *2006 IEEE Intelligent Vehicles Symposium*, pp. 206-212, Jun. 2006.

[7] P. Koonce *et al.*, "Traffic Signal Timing Manual," Kittelson & Associates, Inc., Portland, OR, Rep. FHWA-HOP-08-024, 2008.

[8] R. Bissessar and C. Tonder, "Pedestrian Scramble Crossings – A Tale of Two Cities," 2008.

[9] C. Liao *et al.*, "Development of Mobile Accessible Pedestrian Signals (MAPS) for Blind Pedestrians at Signalized Intersections," Center for Transportation Studies, Minneapolis, MN, Rep. CTS 11-11, Jun. 2011.

[10] R. V. Houten *et al*, "Pedestrian Push-Button Confirmation Increases Call Button Usage and Compliance," *Transportation Research Record: Journal of the Transportation Research Board*, vol. 1982, no. 1, pp. 99-103, Jan. 2006.

[11] J.S. Schiller, J.W. Lucas, B.W. Ward, J.A. Peregoy, "Summary health statistics for U.S. adults: National Health Interview Survey, 2010." Nat. Center for Health Statistics. Vital Health Stat 10(252). 2012.

[12] R. W. Wall *et al.*, "Application of Plug-and-Play Distributed Signal Technology to Traffic Signals," *Transportation Research Board 85th Annual Meeting*, 2006.

[13] A. Huska, "Application of Plug and Play Distributed Sensor Networks to Traffic Control Signals," M.S. Thesis, Dept. Elec. Comput. Eng., Univ. Idaho, Moscow, ID, 2006.

[14] R. W. Wall and B. Johnson, "Application of Plug-and-Play Distributed Signal Technology to Traffic Signals," NIATT, Univ. Idaho, Moscow, ID, Rep. N06-01, Jan. 2006.

[15] Z. Sapp, private communication, Apr. 2014.

[16] *Traffic Control Systems*, NEMA TS 1-1989 ver. R2005, 2004.

[17] *Traffic Controller Assemblies with NTCIP Requirements*, NEMA TS2 ver. 02.06, 2003.

[18] *Type 170 Traffic Signal Controller System–Hardware Specification*, FHWA-IP-78-16, 1978.

[19] *Advanced Transportation Controller (ATC) Standard*, ATC ver. 5.2b, 2006.

[20] RITA and US DOT. (2011, Apr. 7) [Online]. "Deployment Statistics," Available: http://www.itsdeployment.its.dot.gov/am.aspx

[21] Econolite, *NEMA TS2 Standards: The "Intelligent Cabinet"*, Anaheim, CA.

[22] *Synchronous Data Link Control-Concepts*, 4th ed., IBM, Research Triangle Park, NC, GA27-3093-3, 1986.

[23] P. Olson, private communication, Jul. 2014.

[24] *MMU-16LE Series SmartMonitor*, Apr. 2012 ed., EDI Inc., Phoenix, AZ, pp. 30-31.

[25] *Advisor Advanced Accessible Pedestrian Station (AAPS)*, Campbell Co., Boise, ID, 2014.

[26] *MID400-AC Line Monitor Logic-Out Device*, Rev. 1.0.4, Fairchild Semiconductor Co., San Jose, CA, 2010.

[27] Z. Sapp, "Real Time Network Control for Advanced Accessible Pedestrian Systems Using Ethernet Over Power Line," M.S. Thesis, Dept. Elec. Comput. Eng., Univ. Idaho, Moscow, ID, 2010.

[28] C. Browne, "The Evolution of Communication in the Advanced Accessible Pedestrian System," unpublished.

[29] R.W. Wall, "Smart Traffic Signals Applicaiton of Real-Time Distributed Embedded Systems," *WSU Tri-Cities Research Colloquium*, April 2012.

[30] L. A. Klein *et al.*, "Traffic Detector Handbook: Third Edition-Volume II," Federal Highway Administration, McLean, VA, Rep. FHWA-HRT-06-139, Oct. 2006.

[31] D. DeVoe, "Application of Intelligent Transportation Systems Protocols for Controlling a Distributed Network of Advanced Traffic Devices," M.S. Thesis, Dept. Elec. Comput. Eng., Univ. Idaho, Moscow, ID, 2009.

[32] G. DeRuwe, "Pedestrian Assistance Using Distributed Smart Signals Traffic Controls," M.S. Thesis, Dept. Elec. Comput. Eng., Univ. Idaho, Moscow, ID, 2009.

[33] Y. Zhou, "The Design of the NIATT Controller Interface Device," M.S. Thesis, Dept. Elec. Comput. Eng., Univ. Idaho, Moscow, ID, 2000.

[34] M. Soltero *et al.*, "RS-422 and RS-485 Standards Overview and System Configurations," Texas Instruments, Dallas, TX, Rep. SLLA070D, Jun. 2002.

[35] W. Stallings, "Error Detection," in *Data and Computer Communications*, 9th ed. Upper Saddle River: Prentice Hall, 2011, ch. 6, sec. 3, pp. 186-196.

[36] M. Yu *et al.*, "A Secure Architecture for Traffic Control Systems with SDLC Protocols," in *IEEE International Conference on Systems, Man, and Cybernetics*, Seoul, Korea, 2012, pp. 2161-2166.

[37] *MAX3483E/MAX3485E/MAX3486E/MAX3488E/MAX3490E/MAX3491E 3.3V-Powered, ±15kV ESD-Protected, 12Mbps and Slew-Rate-Limited True RS-485/RS-422 Transceivers*, Rev 0, Maxim Integrated, San Jose, CA, 1999.

[38] *MAX3280E/MAX3281E/MAX3283E/MAX3284E ±15kV ESD-Protected 52Mbps, 3V to 5.5V, SOT23 RS-485/RS-422 True Fail-Safe Receivers*, Rev 2, Maxim Integrated, San Jose, CA, 2012.

[39] *MachXO2 Family Handbook*, rev. 03.8, Lattice Semiconductor, Portland, OR, 2013.

[40] *Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, OpenCores, 2010.

[41] *MachXO2 Programming and Configuration Usage Guide*, rev. 3.1, Lattice Semiconductor, Portland, OR, 2014.

[42] R. Billinton and R. Allan, "Network Modelling and Evaluation of Simple Systems," in *Reliability Evaluation of Engineering Systems*, 2nd ed. New York: Plenum Press, 1992, ch. 4, pp. 81-98.

[43] *Military Handbook: Reliability Prediction of Electronic Equipment*, MIL-HDBK-217F, Department of Defense, Washington D.C., 1990.

[44] S. Thorpe *et al.*, "Speed of Processing in the Human Visual System," *Nature*, vol. 381, pp. 520-522, Mar. 1996.

Appendix A

APC Schematics for SDLC Decoding

The following schematics were printed with permission by Richard Wall at the University of Idaho in Moscow, ID for use in the APC. This APC implementation utilizes the NEMA TS2 SDLC network for pedestrian signal decoding. It includes circuitry used in past implementations (shown in Figure A.1 and in Figure A.2) that connect directly to signal load switches for signalized intersections that do not implement the NEMA TS2 standard. A complete parts list is shown in Table A.1 and Table A.2.

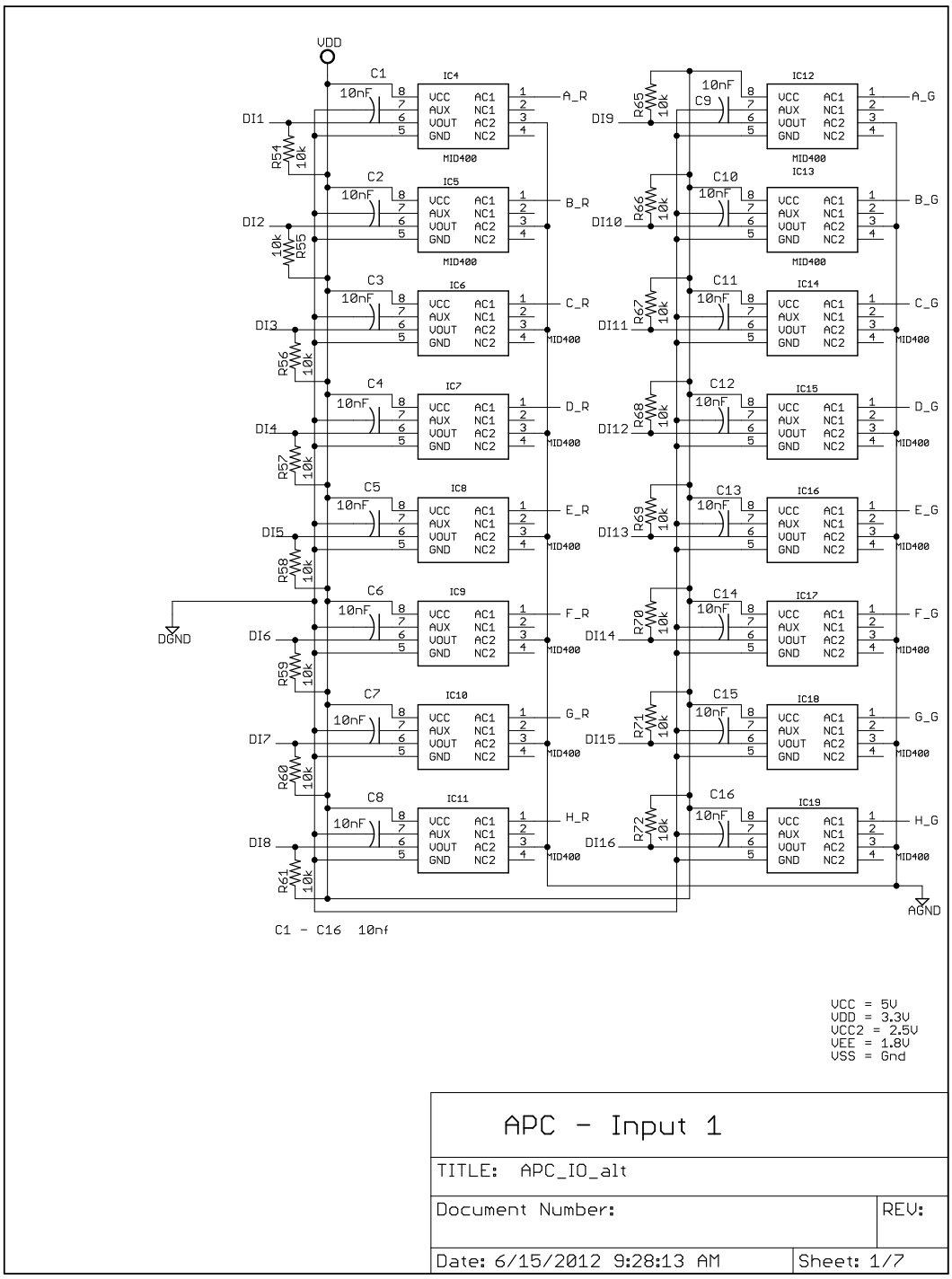

Figure A.1: APC Schematic - Load switch inputs 1.

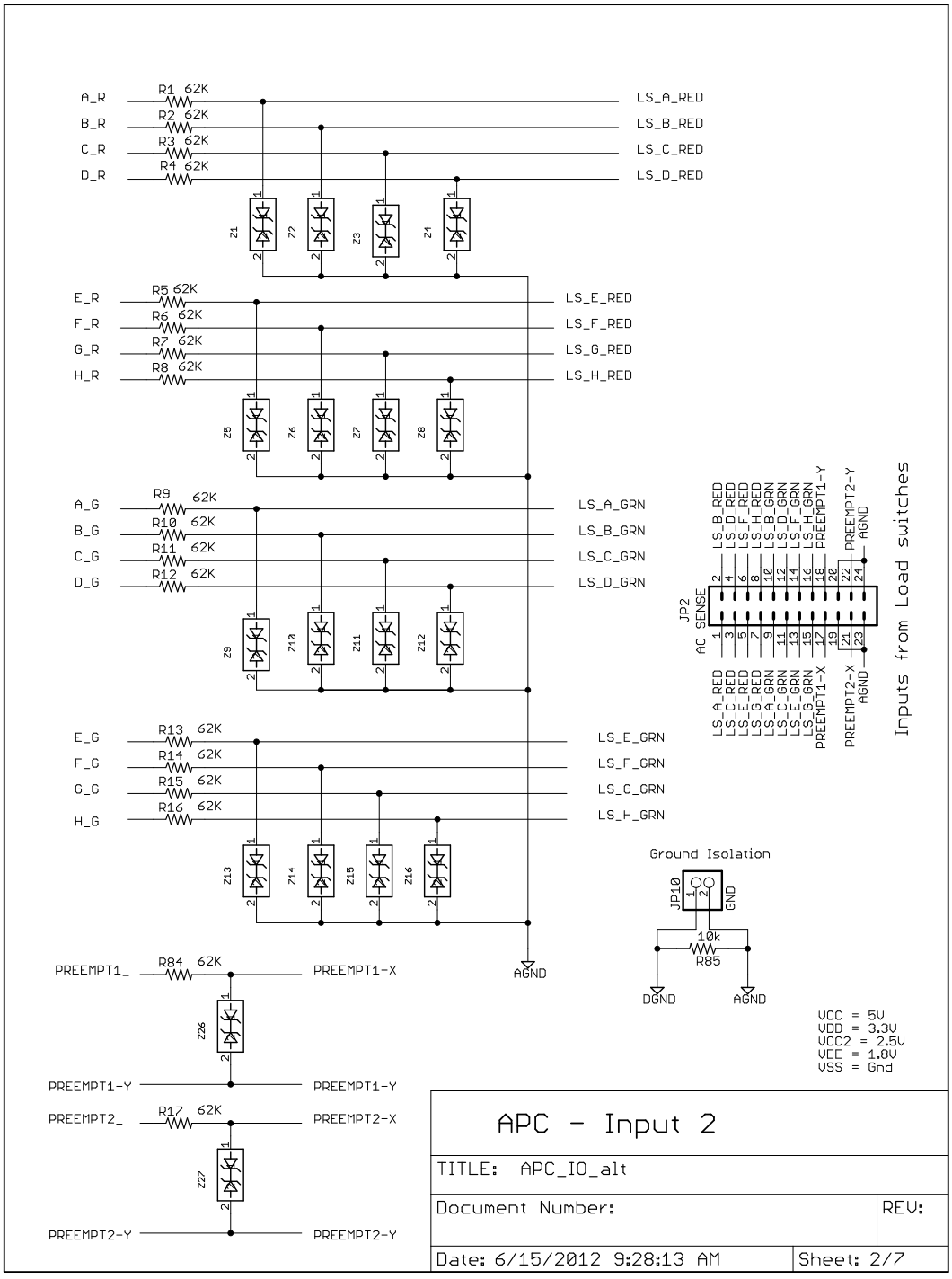

Figure A.2: APC Schematic - Load switch inputs 2.

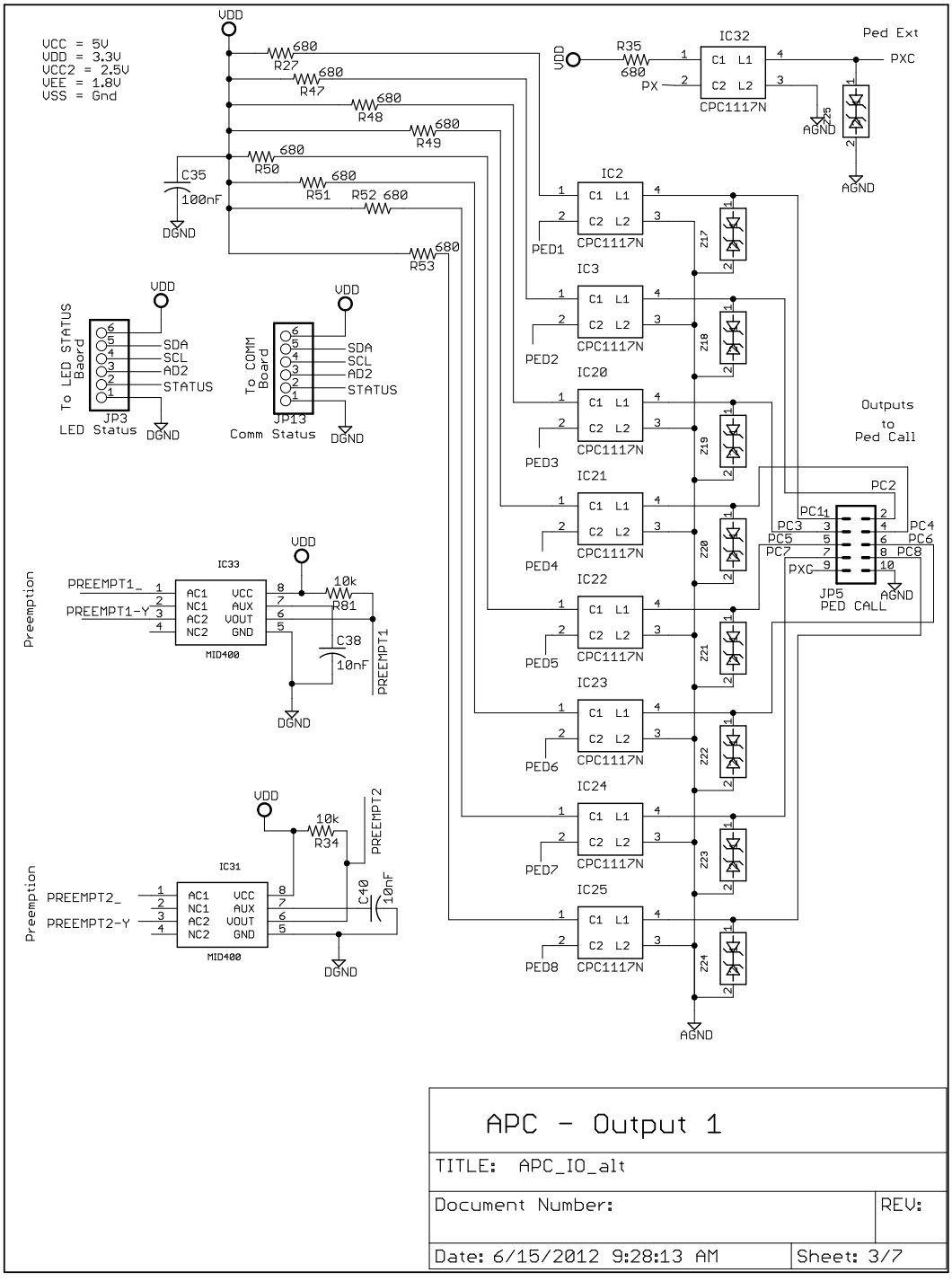

Figure A.3: APC Schematic - Pedestrian call outputs.

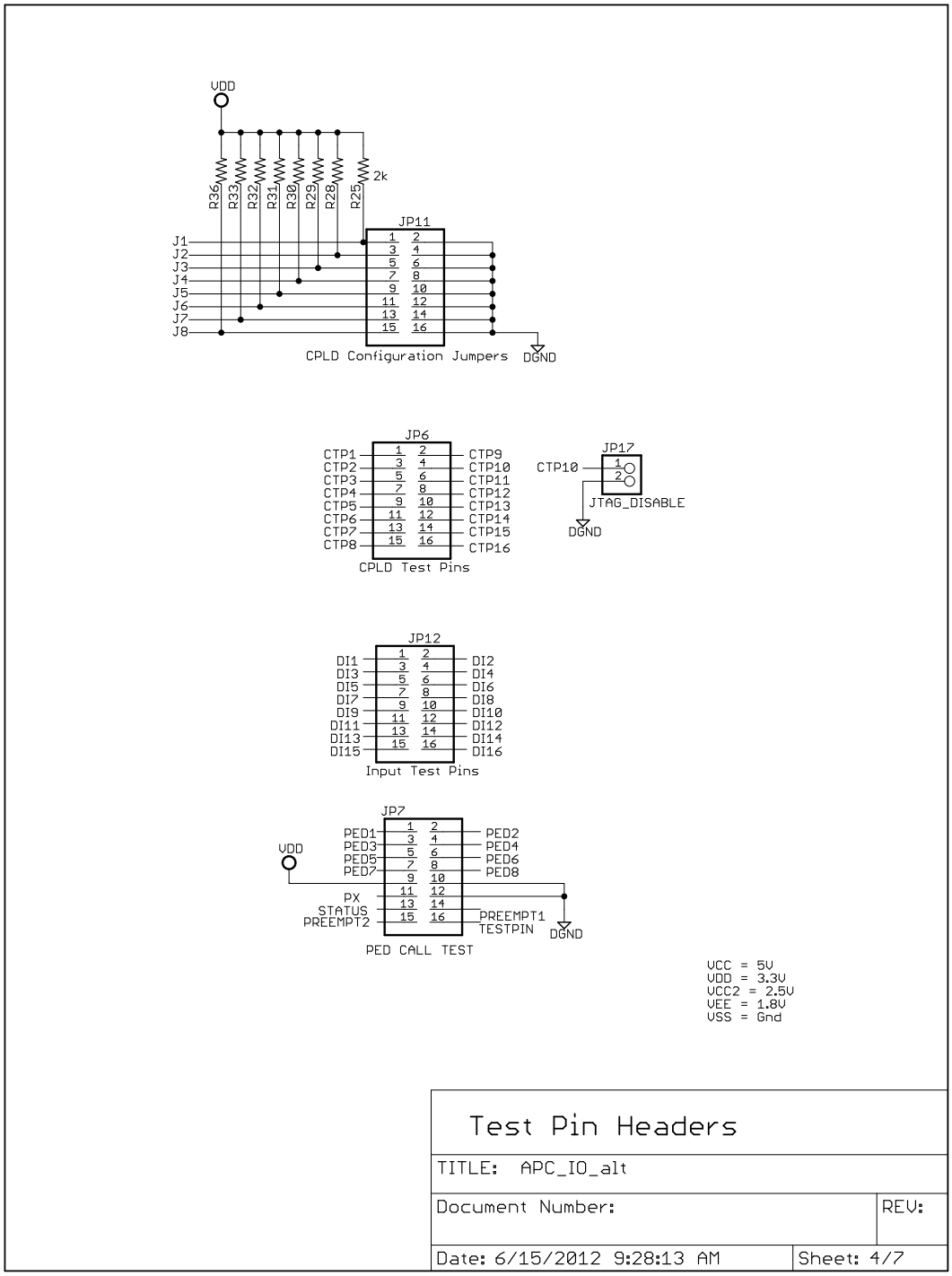

Figure A.4: APC Schematic - Test headers.

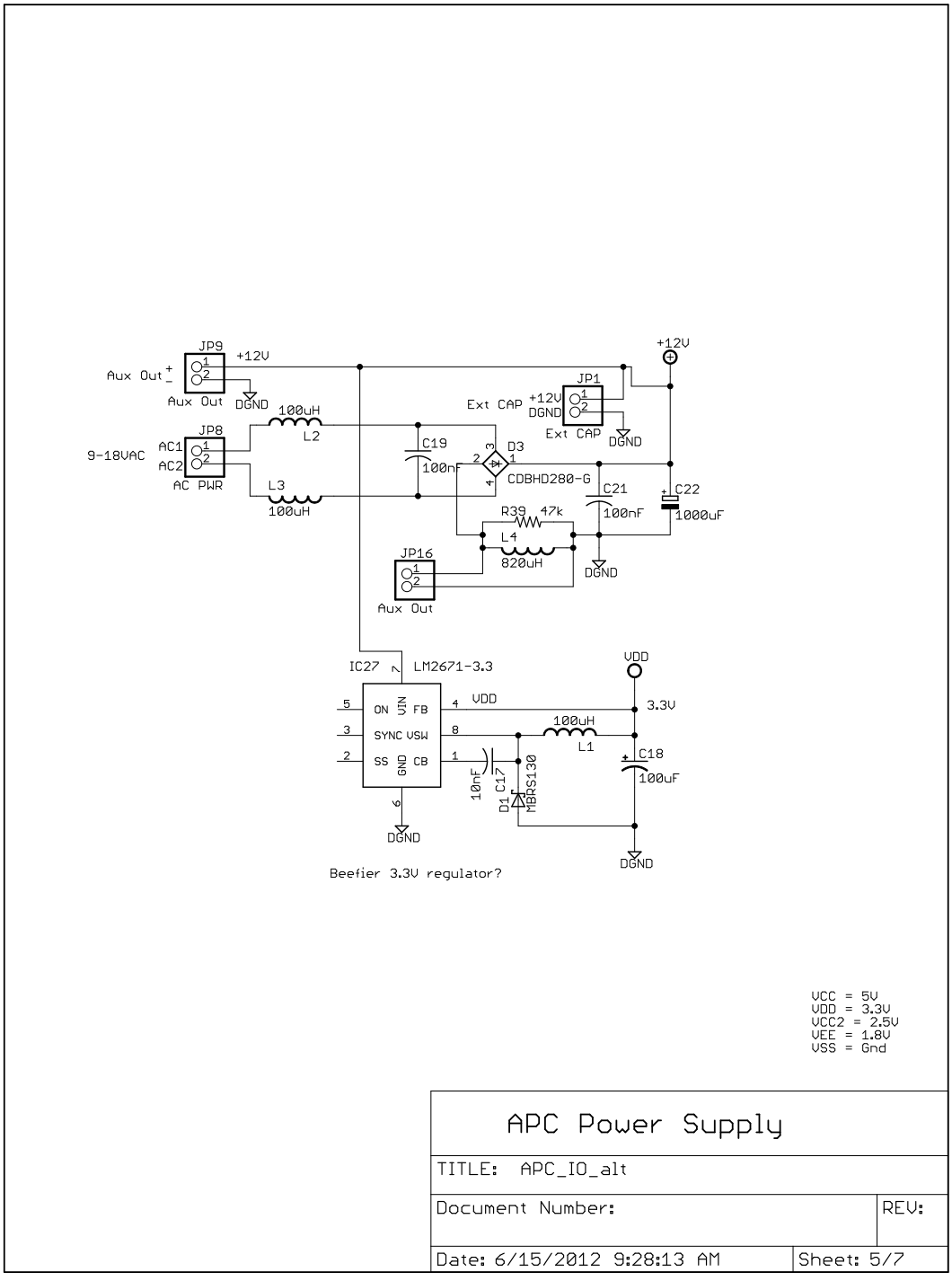

Figure A.5: APC Schematic - Power supply.

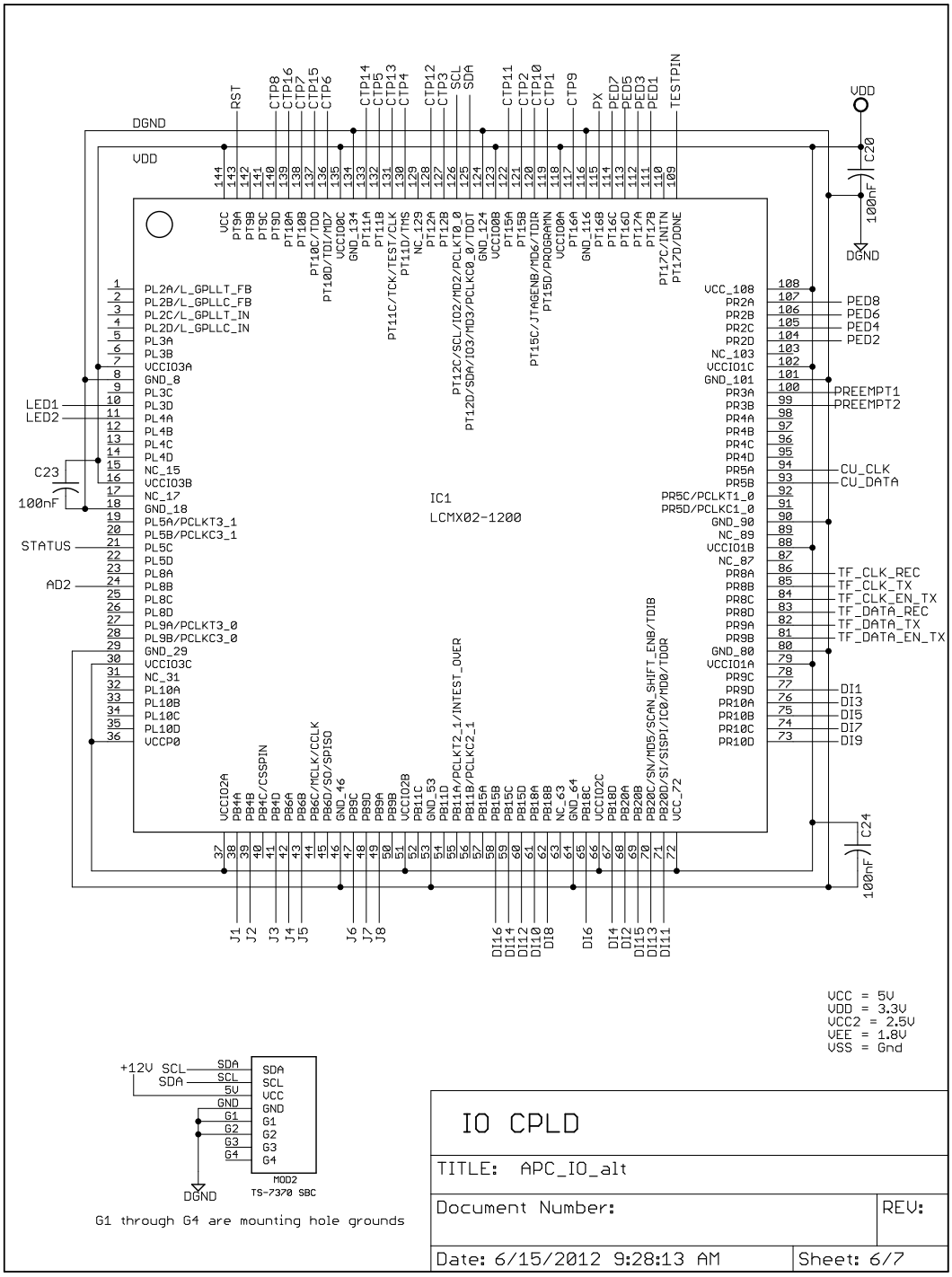

Figure A.6: APC Schematic - CPLD.

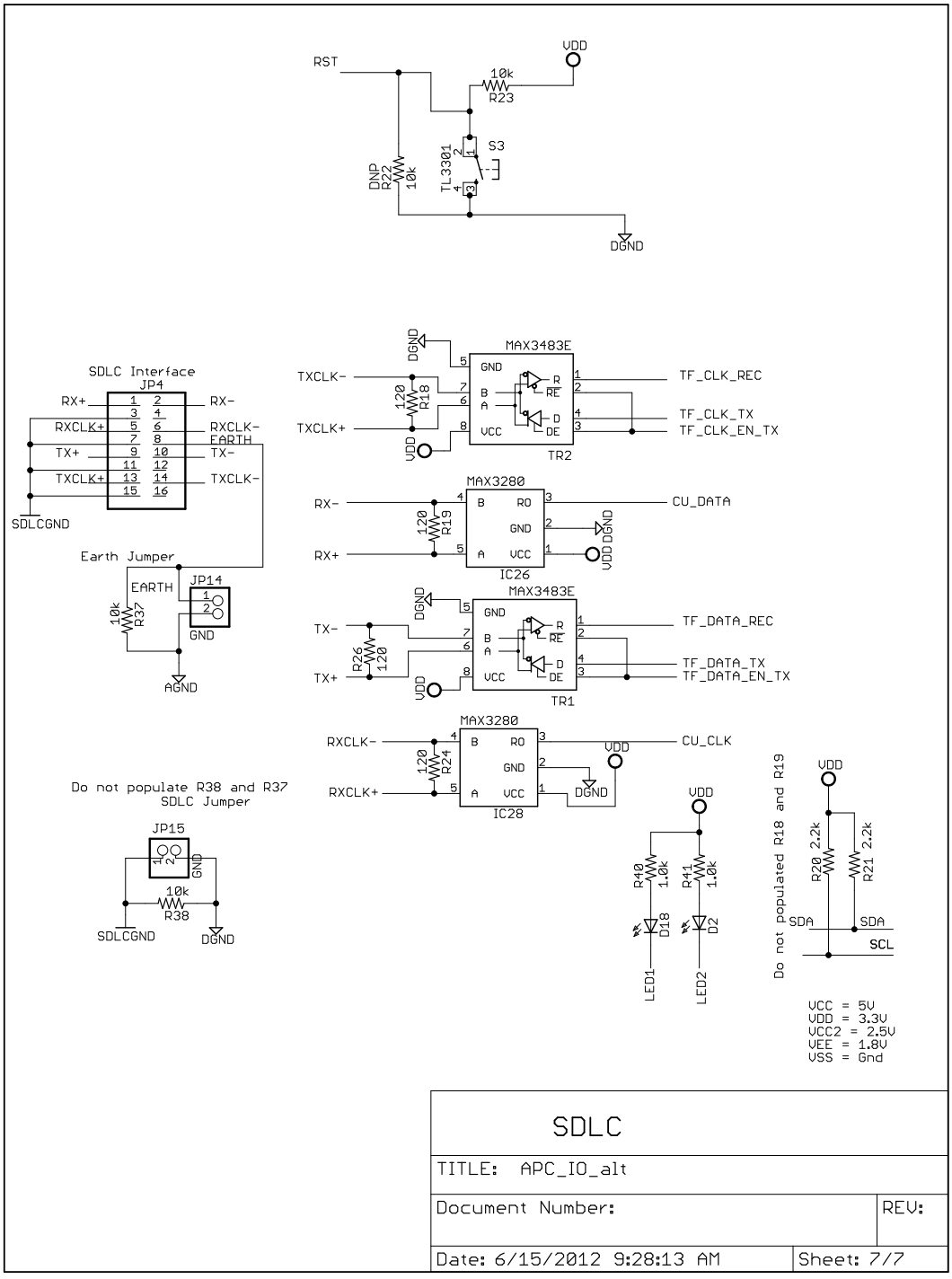

Figure A.7: APC Schematic - SDLC EIA-485 translation.

Component	**Value**	**Manufacturer Part Number**	**Description**
C1-C17, C38, C40	10nF	C3216X7R2J103K	CAP CER 10000PF 630V X7R 1206
C18	100uF	EEE-1VA101P	CAP ALUM 100UF 35V 20% SMD
C19, C21, C35	100nF	C3216X7R1H104K	CAP CER 0.1UF 50V 10% X7R 1206
C20, C23, C24	100nF	C1206F104K3RACTU	CAP CER 0.1UF 25V 10% X7R 1206
C22	100uF	EEV-FK1E102Q	CAP ALUM 1000UF 25V 20% SMD
D1	30V 1A	MBRS130LT3	DIODE SCHOTTKY 30V 1A SMB
D2, D18	570NM	LG N971-KN-1	LED CHIPLED 570NM GREEN 1206 SMD
D3	2A 100V	CDBHD2100-G	DIODE BRIDGE 2A 100V TO-269AA
R1-R17, R84	62kΩ	CRCW251262K0JNEG	RES 62K OHM 1W 5% 2512 SMD
R18, R19, R24, R26	120Ω	CRCW1206120RFKEA	RES 120 OHM 1/4W 1% 1206 SMD
R20, R21	2.2kΩ	CRCW12062K20FKEA	RES 2.20K OHM 1/4W 1% 1206 SMD
R22, R23, R34, R37, R38, R54-R61, R65-R72, R81, R85	10kΩ	CRCW120610K0FKEA	RES 10.0K OHM 1/4W 1% 1206 SMD
R25, R28-R33, R36	2kΩ	CRCW12062K00FKEA	RES 2.00K OHM 1/4W 1% 1206 SMD
R27, R35, R47-R53	680Ω	CRCW1206680RFKEA	RES 680 OHM 1/4W 1% 1206 SMD
R39	47kΩ	CRCW120647K0FKEA	RES 47.0K OHM 1/4W 1% 1206 SMD
R40, R41	1.0kΩ	CRCW12061K00FKEA	RES 1.00K OHM 1/4W 1% 1206 SMD
L1, L2, L3	100uH	SLF7045T-101MR50-PF	INDUCTOR SHIELD PWR 100UH 7045
L4	820uH	DR74-821-R	INDUCTOR SHIELD PWR 820UH SMD
Z1-Z16, Z26, Z27	240V 600A	ERZ-VF2M241	SURGE ABSORBER 240V 600A VFM ZNR
Z17-Z25	30VRMS	CT1210K30G	VARISTOR 30VRMS 1210 SMD

Table A.1: APC parts list–passive components.

Component	**Value**	**Manufacturer Part Number**	**Description**
IC1	1280LUTS 108 I/O	LCMXO2-1200HC-4TG144I	IC PLD 1280LUTS 108I/O 144TQFP
IC2, IC3, IC20-IC25, IC32	150mA	CPC1117NTR	RELAY OPTOMOS 150MA 4-SOP
IC4-IC19, IC31, IC33	2500 VRMS	MID400S	8-PIN OPTOCOUPLER LOGIC AC LINE
IC27	3.3V 500mA	LM2671M-3.3/NOPB	IC REG SIMPLE SWITCHER 8-SOIC
IC26, IC28	RS422, RS485	MAX3280EAUK+T	IC RCVR RS485/422 SOT23-5
TR1, TR2	RS422, RS485	MAX3483EESA+	IC TXRX RS485/422 250KBPS 8-SOIC
S3	0.05A 12V	TL3301NF160QG/TR	SWITCH TACTILE SPST-NO 0.05A 12V

Table A.2: APC parts list–active components.

Appendix B

HDL Listings

Listing B.1: Top module.

```
1  -- USE LCMXO2-1200HC-4TG144I FOR ACTUAL APC I/O BOARD
2  -- USE LCMXO2-1200ZE-1TG144C FOR BREAKOUT BOARD
3
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.all;
6
7  entity top is
8    port(
9          -- Control signals
10         --slic_mclk      : in std_logic; -- internal oscillator used
11         sli_rst          : in std_logic; --active low
12
13         -- I2C Signals
14         slio_sda                 : inout std_logic;
15         slcio_scl        : inout std_logic;
16
17         -- SDLC Signals
18         -- TCOUT Ignore these signals for now.
19         slci_sdlc_tcoutclk       : in std_logic;
20         sli_sdlc_tcoutdata       : in std_logic;
21         slo_sdlc_tcin_txen       : out std_logic;
22         -- TCIN In the future, this may be an output port hence the
               en line
23         slci_sdlc_tcin_clk       : in std_logic;
24         sli_sdlc_tcin_data       : in std_logic;
25
26
27         -- Parallel I/O signals read from MID400s
28         --slvo_calls             : out std_logic_vector( 7 downto 0 );
29         slvi_walks               : in std_logic_vector( 7 downto 0 );
               -- DI9 - DI16
30         slvi_dontwalks : in std_logic_vector( 7 downto 0 ); -- DI1 -
               DI8
31         sli_preempt1 : in std_logic;
32         sli_preempt2 : in std_logic;
33
34         -- Ped call outputs
35         slvo_ped : out std_logic_vector( 7 downto 0 );
36         slo_px : out std_logic;
37
38
39         -- Configuration jumpers
40         slvi_j : in std_logic_vector( 7 downto 0 );
```

```
41
42          -- LED board signals
43          slo_status : out std_logic;
44          slo_ad2 : out std_logic; -- address bit
45
46          -- Test signals
47          --slvo_ctp : in std_logic_vector( 15 downto 0 );
48          sli_ctp1 : out std_logic;
49          sli_ctp2 : in std_logic;
50          sli_ctp3 : in std_logic;
51          sli_ctp4 : out std_logic;
52          sli_ctp5 : out std_logic;
53          sli_ctp6 : in std_logic;
54          sli_ctp7 : out std_logic;
55          sli_ctp8 : in std_logic;
56          sli_ctp9 : out std_logic;
57          sli_ctp10 : in std_logic;
58          sli_ctp11 : in std_logic;
59          sli_ctp12 : in std_logic;
60          sli_ctp13 : in std_logic;
61          sli_ctp14 : in std_logic;
62          sli_ctp15 : in std_logic;
63          sli_ctp16 : in std_logic;
64          -- Info LEDs:
65          -- led1 on = SDLC decoding - led1 off = wiring to field
                terminals decoding
66          -- led2 = same as status LED on LED board
67          slo_led1 : out std_logic; -- pin 10 on APC, pin 97 on
                BREAKOUT
68          slo_led2 : out std_logic; -- pin 11 on APC, pin 98 on
                BREAKOUT
69          slo_testpin : out std_logic
70        );
71  end top;
72
73  architecture Behavioral of top is
74
75    component apc2_efb
76      port (
77              wb_clk_i: in  std_logic;
78              wb_rst_i: in  std_logic;
79              wb_cyc_i: in  std_logic;
80              wb_stb_i: in  std_logic;
81              wb_we_i: in  std_logic;
82              wb_adr_i: in  std_logic_vector(7 downto 0);
83              wb_dat_i: in  std_logic_vector(7 downto 0);
```

```
84              wb_dat_o: out  std_logic_vector(7 downto 0);
85              wb_ack_o: out  std_logic;
86              i2c1_scl: inout  std_logic;
87              i2c1_sda: inout  std_logic;
88              i2c1_irqo: out  std_logic;
89              tc_clki : in std_logic;
90              tc_ic : in std_logic;
91              tc_rstn : in std_logic;
92              tc_int : out std_logic;
93              tc_oc : out std_logic
94            );
95    end component;
96
97
98    component sdlcdecoder
99      port(
100           sli_sdlc_data : in STD_LOGIC;
101           sli_sdlc_clock : in STD_LOGIC;
102           slvo_sdlc_wlk : out STD_LOGIC_VECTOR( 15 downto 0 );
103           slvo_sdlc_dw : out STD_LOGIC_VECTOR( 15 downto 0 );
104           sli_reset : in STD_LOGIC;
105           sli_clk : in STD_LOGIC;
106           slo_crc_pass : out std_logic;
107           slo_sdlc_busy : out std_logic
108         );
109   end component;
110
111
112   component fdwdecoder
113     port (
114             slvi_sdlc_wlk : in std_logic_vector( 15 downto 0 );
115             slvi_sdlc_dw : in std_logic_vector( 15 downto 0 );
116             slvo_sdlc_fdw : out std_logic_vector( 15 downto 0 );
117             sli_reset : in std_logic;
118             sli_clk : in std_logic;
119             sli_clk_div : in std_logic;
120             sli_sdlc_busy : in std_logic
121           );
122   end component;
123
124   -- MASTER WISHBONE STATE MACHINE
125   component wbmaster_fsm is
126     port(
127           sli_clk : in std_logic;
128           sli_reset : in std_logic;
129           wbc_en : out std_logic;
```

```
130             wbc_done : in std_logic;
131             wbc_src : out std_logic_vector( 7 downto 0 );
132             wbc_dest : out std_logic_vector( 7 downto 0 );
133             r0 : in std_logic_vector( 7 downto 0 );
134             r1 : in std_logic_vector( 7 downto 0 );
135             r2 : out std_logic_vector( 7 downto 0 )
136           );
137   end component;
138
139
140   component wbcontroller is
141     port(
142           -- WISHBONE controller signals
143           sli_wb_ack    : in std_logic;
144           slci_wb_clk   : in std_logic;
145           sli_wb_rst    : in std_logic;
146           slo_wb_cyc    : out std_logic;
147           slo_wb_stb    : out std_logic;
148           slo_wb_we     : out std_logic;
149           slvo_wb_addr  : out std_logic_vector( 7 downto 0 );
150           slvi_wb_datain : in  std_logic_vector( 7 downto 0 );
151           slvo_wb_dataout : out std_logic_vector( 7 downto 0 );
152
153           -- NON-WISHBONE Signals
154           sli_en        : in std_logic;
155           slvi_addr_src : in std_logic_vector( 7 downto 0 );
156           slvi_addr_dest : in std_logic_vector( 7 downto 0 );
157           slo_done      : out std_logic
158         );
159   end component;
160
161   component wbslave8_out is
162     port (
163            -- WISHBONE Signals
164            sloz_wb_ack  : out std_logic;
165            slci_wb_clk  : in  std_logic;
166            sli_wb_rst   : in  std_logic;
167            sli_wb_cyc   : in  std_logic;
168            sli_wb_stb   : in  std_logic;
169            sli_wb_we    : in  std_logic;
170            slvi_wb_addr : in  std_logic_vector( 7 downto 0 );
171            slvi_wb_datain : in  std_logic_vector( 7 downto 0 );
172            slvoz_wb_dataout : out std_logic_vector( 7 downto 0 );
173
174            -- NON-WISHBONE Signals
175            slvo_data    : out std_logic_vector( 7 downto 0 );
```

```
176             slvi_addr_set : in std_logic_vector( 7 downto 0 )
177           );
178   end component;
179
180   component wbslave8_in is
181     port (
182             -- WISHBONE Signals
183             sloz_wb_ack                   : out std_logic;
184             --slci_wb_clk                 : in  std_logic;
185             --sli_wb_rst                  : in  std_logic;
186             --sli_wb_cyc                  : in  std_logic;
187             --sli_wb_stb                  : in  std_logic;
188             --sli_wb_we                   : in  std_logic;
189             slvi_wb_addr : in  std_logic_vector( 7 downto 0 );
190             slvoz_wb_dataout : out std_logic_vector( 7 downto 0 );
191
192             -- NON-WISHBONE Signals
193             slvi_data     : in std_logic_vector( 7 downto 0 );
194             slvi_addr_set : in std_logic_vector( 7 downto 0 )
195           );
196   end component;
197
198
199   component conflictcheck is
200     port(
201           sli_clk : in std_logic;
202           sli_clk_div : in std_logic;
203           sli_rst : in std_logic;
204
205           slo_conflict : out std_logic;
206
207           slvi_dw : in std_logic_vector( 15 downto 0 );
208           slvi_w : in std_logic_vector( 15 downto 0 );
209
210           slvo_dw : out std_logic_vector( 15 downto 0 );
211           slvo_w : out std_logic_vector( 15 downto 0 )
212         );
213   end component;
214
215
216   -- Internal oscillator
217   COMPONENT OSCH
218     -- synthesis translate_off
219     GENERIC  (NOM_FREQ: string := "10.23");
220     --GENERIC  (NOM_FREQ: string := "133");
221     -- synthesis translate_on
```

```
222     PORT (
223             STDBY :IN std_logic;
224             OSC :OUT std_logic;
225             SEDSTDBY :OUT std_logic
226           );
227   END COMPONENT;
228
229   attribute NOM_FREQ : string;
230   attribute NOM_FREQ of OSCinst0 : label is "10.23";
231   -- attribute NOM_FREQ of OSCinst0 : label is "133";
232
233
234   -- Intenral oscillator signals
235   signal stdby : std_logic;
236   signal osc_int : std_logic;
237   signal stdby_sed : std_logic;
238   signal sl_mclk : std_logic;
239
240   -- WISHBONE bus signals ( referenced to the master )
241   signal slz_wb_ack     : std_logic;
242   signal slc_wb_clk     : std_logic;
243   signal sl_wb_rst      : std_logic;
244   signal sl_wb_cyc      : std_logic;
245   signal sl_wb_stb      : std_logic;
246   signal sl_wb_we       : std_logic;
247   signal slv_wb_addr    : std_logic_vector( 7 downto 0 );
248   signal slv_wb_dataout : std_logic_vector( 7 downto 0 );
249   signal slvz_wb_datain : std_logic_vector( 7 downto 0 );
250
251   -- WISHBONE controller signals.
252   signal sl_wbc_en      : std_logic;
253   signal sl_wbc_done    : std_logic;
254   signal slv_addr_src   : std_logic_vector( 7 downto 0 );
255   signal slv_addr_dest  : std_logic_vector( 7 downto 0 );
256
257   -- Master state machine signals.
258   signal slv_cnfg       : std_logic_vector( 7 downto 0 );
259   signal slv_r0         : std_logic_vector( 7 downto 0 );
260   signal slv_r1         : std_logic_vector( 7 downto 0 );
261   signal slv_r2         : std_logic_vector( 7 downto 0 );
262
263   -- Ped signal signals
264   signal slv_dw         : std_logic_vector( 15 downto 0 );
265   signal slv_fdw              : std_logic_vector( 15 downto 0 );
266   signal slv_w          : std_logic_vector( 15 downto 0 );
267
```

```
268   -- Signals coming out of the SDLC decoder
269   signal slv_sdlc_dontwalks     : std_logic_vector( 15 downto 0 );
270   signal slv_sdlc_walks         : std_logic_vector( 15 downto 0 );
271   signal sl_sdlc_busy           : std_logic;
272
273   -- EFB signals
274   signal slvz_wb_datain_efb : std_logic_vector( 7 downto 0 );
275   signal slz_wb_ack_efb : std_logic;
276   signal efb_en : std_logic;
277   signal i2c_int : std_logic;
278
279   signal slv_calls : std_logic_vector( 7 downto 0 );
280   signal slv_status : std_logic_vector( 7 downto 0 );
281   signal sl_rst : std_logic;
282
283   signal sl_tc_ic : std_logic;
284   signal sl_tc_rstn : std_logic;
285   signal sl_tc_int : std_logic;
286   signal sl_tc_oc : std_logic;
287
288   -- CRC signals
289   signal sl_crc_pass : std_logic;
290
291   -- I2C busy signal
292   signal sl_i2c_busy : std_logic;
293
294   -- conflict signal
295   signal sl_conflict : std_logic;
296
297   -- RAW SDLC DW signal
298   signal slv_dw_temp1 : std_logic_vector( 15 downto 0 );
299   signal slv_dw_temp0   : std_logic_vector( 15 downto 0 );
300   signal slv_fdw_temp   : std_logic_vector( 15 downto 0 );
301   signal slv_w_temp     : std_logic_vector( 15 downto 0 );
302
303 begin
304
305   -- Setup system clock
306   OSCInst0: OSCH
307   -- synthesis translate_off
308   GENERIC MAP ( NOM_FREQ  => "10.23" )
309   --GENERIC MAP ( NOM_FREQ  => "133" )
310   -- synthesis translate_on
311   PORT MAP (
312               STDBY => stdby  ,
313               OSC => sl_mclk  ,
```

```
314                 SEDSTDBY => stdby_sed
315               );
316
317     -- Set transeivers to receive
318     slo_sdlc_tcin_txen <= '0'; -- change this to a config register bit
319
320     -- PED Call outputs
321     slvo_ped(0) <= slv_calls(0);
322     slvo_ped(1) <= slv_calls(1);
323     slvo_ped(2) <= slv_calls(2);
324     slvo_ped(3) <= slv_calls(3);
325     slvo_ped(4) <= slv_calls(4);
326     slvo_ped(5) <= slv_calls(5);
327     slvo_ped(6) <= slv_calls(6);
328     slvo_ped(7) <= slv_calls(7);
329     slo_px <= slv_cnfg(3);
330
331     -- LED outputs
332     slo_led1 <= slv_cnfg(0);
333     slo_led2 <= slv_status(4);
334     slo_status <= not slv_status(4); -- active low LED on
335     slo_ad2 <= '1';
336
337     slo_testpin <= sl_crc_pass;
338
339     -- I2C busy : slv_r0( 6 ) = '1';  I2C not busy : slv_r0( 6 ) = '0';
340     sl_i2c_busy <= slv_r0( 6 );
341
342     -- Turn internal oscillator on
343     stdby <= '0';
344
345     -- Clock WISHBONE at system clock
346     slc_wb_clk <= sl_mclk;
347
348     -- Reset signals (embedded function resets are active low)
349     sl_wb_rst <= not sli_rst;
350     sl_rst <= not sli_rst;
351     sl_tc_rstn <= sli_rst;
352
353     -- Multiplex the ped signal data choosing either sdlc or parallel
354     -- error code added
355     -- Don't allow DW and FDW to be on at the same time
356     slv_dw_temp1( 0 ) <= '0' when slv_fdw_temp( 0 ) = '1' else
            slv_dw_temp0( 0 );
357     slv_dw_temp1( 1 ) <= '0' when slv_fdw_temp( 1 ) = '1' else
            slv_dw_temp0( 1 );
```

```
358   slv_dw_temp1( 2 ) <= '0' when slv_fdw_temp( 2 ) = '1' else
          slv_dw_temp0( 2 );
359   slv_dw_temp1( 3 ) <= '0' when slv_fdw_temp( 3 ) = '1' else
          slv_dw_temp0( 3 );
360   slv_dw_temp1( 4 ) <= '0' when slv_fdw_temp( 4 ) = '1' else
          slv_dw_temp0( 4 );
361   slv_dw_temp1( 5 ) <= '0' when slv_fdw_temp( 5 ) = '1' else
          slv_dw_temp0( 5 );
362   slv_dw_temp1( 6 ) <= '0' when slv_fdw_temp( 6 ) = '1' else
          slv_dw_temp0( 6 );
363   slv_dw_temp1( 7 ) <= '0' when slv_fdw_temp( 7 ) = '1' else
          slv_dw_temp0( 7 );
364   slv_dw_temp1( 8 ) <= '0' when slv_fdw_temp( 8 ) = '1' else
          slv_dw_temp0( 8 );
365   slv_dw_temp1( 9 ) <= '0' when slv_fdw_temp( 9 ) = '1' else
          slv_dw_temp0( 9 );
366   slv_dw_temp1( 10 ) <= '0' when slv_fdw_temp( 10 ) = '1' else
          slv_dw_temp0( 10 );
367   slv_dw_temp1( 11 ) <= '0' when slv_fdw_temp( 11 ) = '1' else
          slv_dw_temp0( 11 );
368   slv_dw_temp1( 12 ) <= '0' when slv_fdw_temp( 12 ) = '1' else
          slv_dw_temp0( 12 );
369   slv_dw_temp1( 13 ) <= '0' when slv_fdw_temp( 13 ) = '1' else
          slv_dw_temp0( 13 );
370   slv_dw_temp1( 14 ) <= '0' when slv_fdw_temp( 14 ) = '1' else
          slv_dw_temp0( 14 );
371   slv_dw_temp1( 15 ) <= '0' when slv_fdw_temp( 15 ) = '1' else
          slv_dw_temp0( 15 );
372   slv_dw_temp0 <= "00000000" & not slvi_dontwalks when slv_cnfg(0) =
          '1' else
373                   slv_sdlc_dontwalks when slv_cnfg(0) = '0' and
                        sl_sdlc_busy = '0'
374                   and sl_i2c_busy = '0' else slv_dw_temp0;
375
376   slv_w_temp <= "00000000" & not slvi_walks when slv_cnfg(0) = '1'
          else
377                 slv_sdlc_walks when slv_cnfg(0) = '0' and
                      sl_sdlc_busy = '0' and sl_i2c_busy = '0' else
                      slv_w_temp;
378
379   slv_fdw <= slv_fdw_temp; -- when sl_conflict = '0' else slv_fdw;
380
381   -- efb output tristates
382   efb_en <= '1' when slv_wb_addr(7) = '0' else '0';
383   slvz_wb_datain <= slvz_wb_datain_efb when efb_en = '1' else "
          ZZZZZZZZ";
```

```
384   slz_wb_ack <= slz_wb_ack_efb when efb_en = '1' else 'Z';
385
386   -- WISHBONE controller
387   c_wbcontroller : wbcontroller
388   port map (
389             sli_wb_ack => slz_wb_ack ,
390             slci_wb_clk => slc_wb_clk ,
391             slvo_wb_addr => slv_wb_addr ,
392             slo_wb_cyc => sl_wb_cyc ,
393             slvi_wb_datain => slvz_wb_datain ,
394             slvo_wb_dataout => slv_wb_dataout ,
395             sli_wb_rst => sl_rst ,
396             slo_wb_stb => sl_wb_stb ,
397             slo_wb_we => sl_wb_we ,
398             sli_en => sl_wbc_en ,
399             slo_done => sl_wbc_done ,
400             slvi_addr_src => slv_addr_src,
401             slvi_addr_dest => slv_addr_dest
402           );
403
404   -- WISHBONE MASTER FSM
405   c_wbmaster_fsm : wbmaster_fsm
406   port map (
407             sli_clk => slc_wb_clk ,
408             sli_reset => sl_rst ,
409             wbc_en => sl_wbc_en ,
410             wbc_done => sl_wbc_done ,
411             wbc_src => slv_addr_src ,
412             wbc_dest => slv_addr_dest ,
413             r0 => slv_r0 ,
414             r1 => slv_r1 ,
415             r2 => slv_r2
416           );
417
418   -- Decodes incoming sdlc data
419   c_sdlcdecoder : sdlcdecoder
420   port map (
421             sli_sdlc_data => sli_sdlc_tcin_data ,
422             sli_sdlc_clock => slci_sdlc_tcin_clk         ,
423             slvo_sdlc_wlk => slv_sdlc_walks ,
424             slvo_sdlc_dw =>    slv_sdlc_dontwalks ,
425             sli_reset => sl_rst ,
426             sli_clk => sl_mclk ,
427             slo_crc_pass => sl_crc_pass ,
428             slo_sdlc_busy => sl_sdlc_busy
429           );
```

```
430
431     -- Decodes fdw signals
432     c_fdwdecoder : fdwdecoder
433     port map (
434                 slvi_sdlc_wlk => slv_w_temp ,
435                 slvi_sdlc_dw => slv_dw_temp0 ,
436                 slvo_sdlc_fdw => slv_fdw_temp ,
437                 sli_reset => sl_rst ,
438                 sli_clk => sl_mclk ,
439                 sli_clk_div => sl_tc_oc ,
440                 sli_sdlc_busy => sl_sdlc_busy
441               );
442
443     -- Embedded Function Block
444     c_apc2_efb : apc2_efb
445     port map (
446                 wb_clk_i => slc_wb_clk , wb_rst_i => sl_wb_rst ,
447                 wb_cyc_i => sl_wb_cyc , wb_stb_i => sl_wb_stb ,
448                 wb_we_i => sl_wb_we ,
449                 wb_adr_i(7 downto 0) => slv_wb_addr ,
450                 wb_dat_i(7 downto 0) => slv_wb_dataout ,
451                 wb_dat_o(7 downto 0) => slvz_wb_datain_efb ,
452                 wb_ack_o => slz_wb_ack_efb , i2c1_scl => slcio_scl ,
453                 i2c1_sda => slio_sda , i2c1_irqo => i2c_int ,
454                 tc_ic => sl_tc_ic ,
455                 tc_rstn => sl_tc_rstn  ,
456                 tc_int => sl_tc_int , tc_oc => sl_tc_oc    ,
457                 tc_clki => sl_mclk
458               );
459
460     -- check for signal conflicts
461     c_conflictcheck : conflictcheck
462     port map (
463                 sli_clk => sl_mclk ,
464                 sli_clk_div => sl_tc_oc ,
465                 sli_rst => sl_rst ,
466                 slo_conflict => sl_conflict ,
467                 slvi_dw => slv_dw_temp1 ,
468                 slvi_w => slv_w_temp ,
469                 slvo_dw => slv_dw ,
470                 slvo_w => slv_w
471               );
472
473     -- Configuration register WB port.
474     -- DO NOT READ FROM THIS REGISTER OVER I2C, READ FROM 0x88 INSTEAD
          !!!!
```

```
475     c_cnfg_write : wbslave8_out
476     port map (
477                 sloz_wb_ack => slz_wb_ack ,
478                 slvi_wb_addr => slv_wb_addr ,
479                 slci_wb_clk => slc_wb_clk ,
480                 slvi_wb_datain => slv_wb_dataout ,
481                 slvoz_wb_dataout => slvz_wb_datain,
482                 sli_wb_rst => sl_rst ,
483                 sli_wb_cyc => sl_wb_cyc ,
484                 sli_wb_stb => sl_wb_stb ,
485                 sli_wb_we =>       sl_wb_we ,
486
487                 slvi_addr_set => X"80" ,
488                 slvo_data => slv_cnfg
489               );
490
491     -- Calls output WB port
492     -- DO NOT READ FROM THIS REGISTER OVER I2C, READ FROM 0x89 INSTEAD
            !!!!
493     c_call : wbslave8_out
494     port map (
495                 sloz_wb_ack => slz_wb_ack ,
496                 slvi_wb_addr => slv_wb_addr ,
497                 slci_wb_clk => slc_wb_clk ,
498                 slvi_wb_datain => slv_wb_dataout ,
499                 slvoz_wb_dataout => slvz_wb_datain ,
500                 sli_wb_rst => sl_rst ,
501                 sli_wb_cyc => sl_wb_cyc ,
502                 sli_wb_stb => sl_wb_stb ,
503                 sli_wb_we =>       sl_wb_we ,
504
505                 slvi_addr_set => X"81" ,
506                 slvo_data => slv_calls
507               );
508
509     -- Temp register WB ports
510     -- R0 is only used to read I2C status register
511     c_r0 : wbslave8_out
512     port map (
513                 sloz_wb_ack => slz_wb_ack ,
514                 slvi_wb_addr => slv_wb_addr ,
515                 slci_wb_clk => slc_wb_clk ,
516                 slvi_wb_datain => slv_wb_dataout ,
517                 slvoz_wb_dataout => slvz_wb_datain ,
518                 sli_wb_rst => sl_rst ,
519                 sli_wb_cyc => sl_wb_cyc ,
```

```
520             sli_wb_stb => sl_wb_stb ,
521             sli_wb_we => sl_wb_we ,
522
523             slvi_addr_set => X"FF" ,
524             slvo_data => slv_r0
525           );
526   c_r1 : wbslave8_out
527   port map (
528             sloz_wb_ack => slz_wb_ack ,
529             slvi_wb_addr => slv_wb_addr ,
530             slci_wb_clk => slc_wb_clk ,
531             slvi_wb_datain => slv_wb_dataout ,
532             slvoz_wb_dataout => slvz_wb_datain ,
533             sli_wb_rst => sl_rst ,
534             sli_wb_cyc => sl_wb_cyc ,
535             sli_wb_stb => sl_wb_stb ,
536             sli_wb_we =>       sl_wb_we ,
537
538             slvi_addr_set => X"FE" ,
539             slvo_data => slv_r1
540           );
541   c_r2 : wbslave8_in
542   port map (
543             sloz_wb_ack => slz_wb_ack ,
544             slvi_wb_addr => slv_wb_addr ,
545                             --slci_wb_clk => slc_wb_clk ,
546                             --slvi_wb_datain => slv_wb_dataout ,
547             slvoz_wb_dataout => slvz_wb_datain ,
548                             --sli_wb_rst => sl_wb_rst ,
549                             --sli_wb_cyc => sl_wb_cyc ,
550                             --sli_wb_stb => sl_wb_stb ,
551                             --sli_wb_we => sl_wb_we ,
552
553             slvi_addr_set => X"FD" ,
554             slvi_data => slv_r2( 7 downto 0 )
555           );
556
557   -- Signal status WB ports
558   c_dwl : wbslave8_in
559   port map (
560             sloz_wb_ack => slz_wb_ack ,
561             slvi_wb_addr => slv_wb_addr ,
562                             --slci_wb_clk => slc_wb_clk ,
563                             --slvi_wb_datain => slv_wb_dataout ,
564             slvoz_wb_dataout => slvz_wb_datain ,
565                             --sli_wb_rst => sl_wb_rst ,
```

```
566                                 --sli_wb_cyc => sl_wb_cyc ,
567                                 --sli_wb_stb => sl_wb_stb ,
568                                 --sli_wb_we => sl_wb_we ,
569
570             slvi_addr_set => X"82" ,
571             slvi_data => slv_dw( 7 downto 0 )
572           );
573   c_dwh : wbslave8_in
574   port map (
575             sloz_wb_ack => slz_wb_ack ,
576             slvi_wb_addr => slv_wb_addr ,
577                                 --slci_wb_clk => slc_wb_clk ,
578                                 --slvi_wb_datain => slv_wb_dataout ,
579             slvoz_wb_dataout => slvz_wb_datain ,
580                                 --sli_wb_rst => sl_wb_rst ,
581                                 --sli_wb_cyc => sl_wb_cyc ,
582                                 --sli_wb_stb => sl_wb_stb ,
583                                 --sli_wb_we => sl_wb_we ,
584
585             slvi_addr_set => X"83" ,
586             slvi_data => slv_dw( 15 downto 8 )
587           );
588   c_fdwl : wbslave8_in
589   port map (
590             sloz_wb_ack => slz_wb_ack ,
591             slvi_wb_addr => slv_wb_addr ,
592                                  --slci_wb_clk => slc_wb_clk ,
593                                  --slvi_wb_datain => slv_wb_dataout
                                        ,
594             slvoz_wb_dataout => slvz_wb_datain ,
595                                  --sli_wb_rst => sl_wb_rst ,
596                                  --sli_wb_cyc => sl_wb_cyc ,
597                                  --sli_wb_stb => sl_wb_stb ,
598                                  --sli_wb_we => sl_wb_we ,
599
600             slvi_addr_set => X"84" ,
601             slvi_data => slv_fdw( 7 downto 0 )
602           );
603   c_fdwh : wbslave8_in
604   port map (
605             sloz_wb_ack => slz_wb_ack ,
606             slvi_wb_addr => slv_wb_addr ,
607                                  --slci_wb_clk => slc_wb_clk ,
608                                  --slvi_wb_datain => slv_wb_dataout
                                        ,
609             slvoz_wb_dataout => slvz_wb_datain ,
```

```
610                                     --sli_wb_rst => sl_wb_rst ,
611                                     --sli_wb_cyc => sl_wb_cyc ,
612                                     --sli_wb_stb => sl_wb_stb ,
613                                     --sli_wb_we => sl_wb_we ,
614
615             slvi_addr_set => X"85" ,
616             slvi_data => slv_fdw( 15 downto 8 )
617           );
618   c_wl : wbslave8_in
619   port map (
620             sloz_wb_ack => slz_wb_ack ,
621             slvi_wb_addr => slv_wb_addr ,
622                             --slci_wb_clk => slc_wb_clk ,
623                             --slvi_wb_datain => slv_wb_dataout ,
624             slvoz_wb_dataout => slvz_wb_datain ,
625                             --sli_wb_rst => sl_wb_rst ,
626                             --sli_wb_cyc => sl_wb_cyc ,
627                             --sli_wb_stb => sl_wb_stb ,
628                             --sli_wb_we => sl_wb_we ,
629
630             slvi_addr_set => X"86" ,
631             slvi_data => slv_w( 7 downto 0 )
632           );
633   c_wh : wbslave8_in
634   port map (
635             sloz_wb_ack => slz_wb_ack ,
636             slvi_wb_addr => slv_wb_addr ,
637                             --slci_wb_clk => slc_wb_clk ,
638                             --slvi_wb_datain => slv_wb_dataout ,
639             slvoz_wb_dataout => slvz_wb_datain ,
640                             --sli_wb_rst => sl_wb_rst ,
641                             --sli_wb_cyc => sl_wb_cyc ,
642                             --sli_wb_stb => sl_wb_stb ,
643                             --sli_wb_we =>  sl_wb_we ,
644
645             slvi_addr_set => X"87" ,
646             slvi_data => slv_w( 15 downto 8 )
647           );
648   c_cnfg_read : wbslave8_in
649   port map ( -- configuration read register
650             sloz_wb_ack => slz_wb_ack ,
651             slvi_wb_addr => slv_wb_addr ,
652             slvoz_wb_dataout => slvz_wb_datain ,
653             slvi_addr_set => X"88" ,
654             slvi_data => slv_cnfg
655           );
```

```
656     c_call_read : wbslave8_in
657     port map ( -- calls read register
658                sloz_wb_ack => slz_wb_ack ,
659                slvi_wb_addr => slv_wb_addr ,
660                slvoz_wb_dataout => slvz_wb_datain ,
661                slvi_addr_set => X"89" ,
662                slvi_data => slv_calls
663              );
664
665     -- Status register
666     -- bit 4: Status LED
667     -- DO NOT READ FROM THIS REGISTER OVER I2C, READ FROM 0x99 INSTEAD
          !!!!
668     c_status : wbslave8_out
669     port map (
670                sloz_wb_ack => slz_wb_ack ,
671                slvi_wb_addr => slv_wb_addr ,
672                slci_wb_clk => slc_wb_clk ,
673                slvi_wb_datain => slv_wb_dataout ,
674                slvoz_wb_dataout => slvz_wb_datain ,
675                sli_wb_rst => sl_rst ,
676                sli_wb_cyc => sl_wb_cyc ,
677                sli_wb_stb => sl_wb_stb ,
678                sli_wb_we =>       sl_wb_we ,
679
680                slvi_addr_set => X"90" ,
681                slvo_data => slv_status
682              );
683     c_status_read : wbslave8_in
684     port map ( -- status read register
685                sloz_wb_ack => slz_wb_ack ,
686                slvi_wb_addr => slv_wb_addr ,
687                slvoz_wb_dataout => slvz_wb_datain ,
688                slvi_addr_set => X"98" ,
689                slvi_data => slv_status
690              );
691
692   end Behavioral;
```

Listing B.2: SDLC decoder module.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity sdlcdecoder is
5    port (
6           sli_sdlc_data : in std_logic;
7           sli_sdlc_clock : in std_logic;
8
9           slvo_sdlc_wlk : out std_logic_vector( 15 downto 0 );
10          --slvo_sdlc_fdw : out std_logic_vector( 15 downto 0 ); --
                jwp
11          slvo_sdlc_dw : out std_logic_vector( 15 downto 0 );
12
13          sli_reset : in std_logic;
14          --flag_out : out std_logic; -- temp
15          sli_clk : in std_logic;
16
17          slo_crc_pass : out std_logic;
18
19          slo_sdlc_busy : out std_logic
20
21
22
23        --sli_clk_out : out std_logic -- tmp
24        );
25 end sdlcdecoder;
26
27 architecture Behavioral of sdlcdecoder is
28
29   signal sl_sdlcclkpulse : std_logic;
30   signal sl_sdlcdata : std_logic;
31   signal sl_sdlcclkpulsenobs : std_logic;
32   signal slv_byte : std_logic_vector( 7 downto 0 );
33   signal slv_poi_address : std_logic_vector( 7 downto 0 );
34   signal slv_poi_control : std_logic_vector( 7 downto 0 );
35   signal slv_poi_frame : std_logic_vector( 7 downto 0 );
36   signal slv_boi : std_logic_vector( 31 downto 0 );
37   signal sl_dse : std_logic;
38   signal sl_bse : std_logic;
39   signal sl_8bits : std_logic;
40   signal sl_flag : std_logic;
41   signal sl_poi : std_logic;
42   signal sl_resetbitcount : std_logic;
```

```
43    signal sl_clearpacket : std_logic;
44    signal sl_saveboi : std_logic;
45    signal sl_push : std_logic;
46    signal sl_crc_pass : std_logic;
47
48    --signal slv_sdlc_fdw : std_logic_vector( 15 downto 0 ); -- jwp
49
50    signal slv_boiout : std_logic_vector( 31 downto 0 );
51
52    component count8 is
53      port (
54             sli_tocount : in std_logic;
55             slo_count8 : out std_logic;
56             sli_reset : in std_logic;
57             sli_clk : in std_logic
58           );
59    end component;
60
61    component risingedge_to_pulse
62      port(
63            sli_edge : in std_logic;
64            sli_clk : in std_logic;
65            sli_reset : in std_logic;
66            slo_pulse : out std_logic
67          );
68    end component;
69
70    component siporeg8
71      port(
72            sli_clk : in std_logic;
73            sli_din : in std_logic;
74            sli_en : in std_logic;
75            sli_reset : in std_logic;
76            slvo_data : out std_logic_vector( 7 downto 0 )
77          );
78    end component;
79
80    component sdlccrc is
81      port (
82             sli_sdlc_data : in std_logic;
83             sli_clock : in std_logic;
84             sli_crc_en : in std_logic;
85
86             sli_reset : in std_logic;
87             sli_clearcrc : in std_logic;
88
```

```
89            slo_crc_pass : out std_logic
90          );
91    end component;
92
93    component destuffer
94      port (
95            sli_data : in std_logic;
96            sli_clockpulse : in std_logic;
97            sli_en : in std_logic;
98
99            slo_data : out std_logic;
100           slo_clockpulse : out std_logic;
101           slo_bse : out std_logic;
102
103           sli_clk : in std_logic;
104           sli_reset : in std_logic
105         );
106   end component;
107
108   component flagdetector
109     port (
110           sli_data : in std_logic;
111           sli_clockpulse : in std_logic;
112           slo_flag : out std_logic;
113           sli_reset : in std_logic;
114           sli_clk : in std_logic
115         );
116   end component;
117
118   component packetbuffer
119     port (
120           slvi_data : in std_logic_vector( 7 downto 0 );
121           sli_push : in std_logic;
122           sli_reset : in std_logic;
123           slvo_address : out std_logic_vector( 7 downto 0 );
124           slvo_control : out std_logic_vector( 7 downto 0 );
125           slvo_frame : out std_logic_vector( 7 downto 0 );
126           slvo_boi : out std_logic_vector( 31 downto 0 );
127           sli_clk : in std_logic
128         );
129   end component;
130
131   component packetcheck
132     port (
133           slvi_poi_address : in std_logic_vector( 7 downto 0 );
134           slvi_poi_control : in std_logic_vector( 7 downto 0 );
```

```
135             slvi_poi_frame : in std_logic_vector( 7 downto 0 );
136             slo_poi : out std_logic
137           );
138   end component;
139
140   component piporeg32
141     port (
142             slvi_data : in std_logic_vector( 31 downto 0 );
143             slvo_data : out std_logic_vector( 31 downto 0 );
144             sli_en : in std_logic;
145             sli_reset : in std_logic;
146             sli_clk : in std_logic
147           );
148   end component;
149
150   component sdlcfsm
151     port (
152             sli_bse : in std_logic;
153             sli_8bits : in std_logic;
154             sli_flag : in std_logic;
155             sli_poi : in std_logic;
156             sli_crc_pass : in std_logic;
157             slo_dse : out std_logic;
158             slo_resetbitcount : out std_logic;
159             slo_clearpacket : out std_logic;
160             slo_saveboi : out std_logic;
161             slo_push : out std_logic;
162             sli_reset : in std_logic;
163             sli_clk : in std_logic
164           );
165   end component;
166
167 begin
168
169   slo_sdlc_busy <= sl_saveboi;
170
171   slvo_sdlc_dw( 15 ) <= slv_boiout( 0 );
172   slvo_sdlc_dw( 14 ) <= slv_boiout( 1 );
173   slvo_sdlc_dw( 13 ) <= slv_boiout( 2 );
174   slvo_sdlc_dw( 12 ) <= slv_boiout( 3 );
175   slvo_sdlc_dw( 11 ) <= slv_boiout( 4 );
176   slvo_sdlc_dw( 10 ) <= slv_boiout( 5 );
177   slvo_sdlc_dw( 9 ) <= slv_boiout( 6 );
178   slvo_sdlc_dw( 8 ) <= slv_boiout( 7 );
179   slvo_sdlc_dw( 7 ) <= slv_boiout( 8 );
180   slvo_sdlc_dw( 6 ) <= slv_boiout( 9 );
```

```
181     slvo_sdlc_dw( 5 ) <= slv_boiout( 10 );
182     slvo_sdlc_dw( 4 ) <= slv_boiout( 11 );
183     slvo_sdlc_dw( 3 ) <= slv_boiout( 12 );
184     slvo_sdlc_dw( 2 ) <= slv_boiout( 13 );
185     slvo_sdlc_dw( 1 ) <= slv_boiout( 14 );
186     slvo_sdlc_dw( 0 ) <= slv_boiout( 15 );
187
188     slvo_sdlc_wlk( 15 ) <= slv_boiout( 16 );
189     slvo_sdlc_wlk( 14 ) <= slv_boiout( 17 );
190     slvo_sdlc_wlk( 13 ) <= slv_boiout( 18 );
191     slvo_sdlc_wlk( 12 ) <= slv_boiout( 19 );
192     slvo_sdlc_wlk( 11 ) <= slv_boiout( 20 );
193     slvo_sdlc_wlk( 10 ) <= slv_boiout( 21 );
194     slvo_sdlc_wlk( 9 ) <= slv_boiout( 22 );
195     slvo_sdlc_wlk( 8 ) <= slv_boiout( 23 );
196     slvo_sdlc_wlk( 7 ) <= slv_boiout( 24 );
197     slvo_sdlc_wlk( 6 ) <= slv_boiout( 25 );
198     slvo_sdlc_wlk( 5 ) <= slv_boiout( 26 );
199     slvo_sdlc_wlk( 4 ) <= slv_boiout( 27 );
200     slvo_sdlc_wlk( 3 ) <= slv_boiout( 28 );
201     slvo_sdlc_wlk( 2 ) <= slv_boiout( 29 );
202     slvo_sdlc_wlk( 1 ) <= slv_boiout( 30 );
203     slvo_sdlc_wlk( 0 ) <= slv_boiout( 31 );
204
205     slo_crc_pass <= sl_crc_pass;
206
207     c_edgetopulse : risingedge_to_pulse
208     port map (
209                 sli_edge => sli_sdlc_clock ,
210                 sli_clk => sli_clk ,
211                 sli_reset => sli_reset ,
212                 slo_pulse => sl_sdlcclkpulse
213               );
214
215     c_flagdetector : flagdetector
216     port map (
217                 sli_data => sli_sdlc_data ,
218                 sli_clockpulse => sl_sdlcclkpulse ,
219                 slo_flag => sl_flag ,
220                 sli_reset => sli_reset ,
221                 sli_clk => sli_clk
222               );
223
224     c_destuffer : destuffer0
225     port map (
226                 sli_data => sli_sdlc_data ,
```

```
227             sli_clockpulse => sl_sdlcclkpulse ,
228             sli_en => sl_dse ,
229             slo_data => sl_sdlcdata ,
230             slo_clockpulse => sl_sdlcclkpulsenobs ,
231             slo_bse => sl_bse ,
232             sli_clk => sli_clk ,
233             sli_reset => sli_reset
234           );
235
236   -- CRC computation
237   c_sdlccrc : sdlccrc
238   port map (
239             sli_sdlc_data => sl_sdlcdata ,
240             sli_clock => sli_clk ,
241             sli_crc_en => sl_sdlcclkpulsenobs ,
242             sli_reset => sli_reset ,
243             sli_clearcrc => sl_clearpacket ,
244             slo_crc_pass => sl_crc_pass
245           );
246
247   c_inbyte : siporeg8
248   port map (
249             sli_clk => sli_clk ,
250             sli_din => sl_sdlcdata ,
251             sli_en => sl_sdlcclkpulsenobs ,
252             sli_reset => sli_reset ,
253             slvo_data => slv_byte
254           );
255
256   c_packetbuffer : packetbuffer
257   port map (
258             slvi_data => slv_byte ,
259             sli_push => sl_push ,
260             sli_reset => sl_clearpacket ,
261             slvo_address => slv_poi_address ,
262             slvo_control => slv_poi_control ,
263             slvo_frame => slv_poi_frame ,
264             slvo_boi => slv_boi ,
265             sli_clk => sli_clk
266           );
267
268   c_packetcheck : packetcheck
269   port map (
270             slvi_poi_address => slv_poi_address ,
271             slvi_poi_control => slv_poi_control ,
272             slvi_poi_frame => slv_poi_frame ,
```

```
273               slo_poi => sl_poi
274             );
275
276     c_boireg : piporeg32
277     port map (
278               slvi_data => slv_boi ,
279               slvo_data => slv_boiout ,
280               sli_en => sl_saveboi ,
281               sli_reset => sli_reset ,
282               sli_clk => sli_clk
283             );
284
285     c_bitcount : count8
286     port map (
287               sli_tocount => sl_sdlcclkpulsenobs ,
288               slo_count8 => sl_8bits ,
289               sli_reset => sl_resetbitcount ,
290               sli_clk => sli_clk
291             );
292
293     c_sdlcfsm : sdlcfsm
294     port map (
295               sli_bse => sl_bse ,
296               sli_8bits => sl_8bits ,
297               sli_flag => sl_flag ,
298               sli_poi => sl_poi ,
299               sli_crc_pass => sl_crc_pass ,
300               slo_dse => sl_dse ,
301               slo_resetbitcount => sl_resetbitcount ,
302               slo_clearpacket => sl_clearpacket ,
303               slo_saveboi => sl_saveboi ,
304               slo_push => sl_push ,
305               sli_reset => sli_reset ,
306               sli_clk => sli_clk
307             );
308   end Behavioral;
```

Listing B.3: Flashing Don't Walk decoder.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity fdwdecoder is
6    port (
7           slvi_sdlc_wlk : in std_logic_vector( 15 downto 0 );
8           slvi_sdlc_dw : in std_logic_vector( 15 downto 0 );
9           slvo_sdlc_fdw : out std_logic_vector( 15 downto 0 );
10          sli_reset : in std_logic;
11          sli_clk : in std_logic; -- system clock
12          sli_clk_div : in std_logic; -- 100ms period clock
13          sli_sdlc_busy : in std_logic
14        );
15 end fdwdecoder;
16
17 architecture Behavioral of fdwdecoder is
18
19   signal dw_sum : std_logic_vector( 15 downto 0 );
20   signal wlk_sum : std_logic_vector( 15 downto 0 );
21   signal dw_product : std_logic_vector( 15 downto 0 );
22   signal wlk_product : std_logic_vector( 15 downto 0 );
23
24   signal sample_start : std_logic;
25   signal sr_en : std_logic;
26   --signal sli_clk_div : std_logic; -- divided clock
27
28   signal fdw : std_logic_vector( 15 downto 0 );
29   signal both : std_logic_vector( 31 downto 0 ); -- holds both wlk
       (15-0) and dw(31-16) signals
30
31   type fdwstate is ( IDLE , WAITCC , GET_DATA );
32   signal fsmstate : fdwstate;
33
34   component fdwcheck
35     port (
36            sli_clk : in std_logic;
37            sli_reset : in std_logic;
38            d_in : in std_logic;
39            sr_en : in std_logic;
40            sum : out std_logic;
41            product : out std_logic
42          );
```

```
43    end component;
44
45    --component clkdivider
46    --port (
47    --sli_clk : in std_logic;
48    --sli_clk_div : out std_logic;
49    --sli_reset : std_logic;
50    --divide_val : std_logic_vector( 31 downto 0 )
51    --);
52    --end component;
53
54    component risingedge_to_pulse
55      port(
56            sli_edge : in std_logic;
57            sli_clk : in std_logic;
58            sli_reset : in std_logic;
59            slo_pulse : out std_logic
60          );
61    end component;
62
63  begin
64
65    -- fsm next state logic
66    p_fsm : process( sli_clk )
67    begin
68      if( rising_edge( sli_clk ) ) then
69        if( sli_reset = '1' ) then
70          fsmstate <= IDLE;
71        else
72          case fsmstate is
73            when IDLE =>
74              if( sample_start = '1' ) then
75                fsmstate <= WAITCC;
76              else
77                fsmstate <= IDLE;
78              end if;
79            when WAITCC =>  -- wait 1 system cc to sample data
80              fsmstate <= GET_DATA;
81            when GET_DATA =>
82              fsmstate <= IDLE;
83          end case;
84
85        end if; -- reset
86      end if; -- rising edge
87    end process;
88
```

```
89   -- internal state logic
90   p_outputs : process( fsmstate )
91   begin
92     case fsmstate is
93       when GET_DATA =>
94         sr_en <= '1';
95       when others =>
96         sr_en <= '0';
97     end case;
98   end process;
99
100  --slvo_sdlc_fdw(15) <= fdw(0);
101  --slvo_sdlc_fdw(14) <= fdw(1);
102  --slvo_sdlc_fdw(13) <= fdw(2);
103  --slvo_sdlc_fdw(12) <= fdw(3);
104  --slvo_sdlc_fdw(11) <= fdw(4);
105  --slvo_sdlc_fdw(10) <= fdw(5);
106  --slvo_sdlc_fdw(9) <= fdw(6);
107  --slvo_sdlc_fdw(8) <= fdw(7);
108  --slvo_sdlc_fdw(7) <= fdw(8);
109  --slvo_sdlc_fdw(6) <= fdw(9);
110  --slvo_sdlc_fdw(5) <= fdw(10);
111  --slvo_sdlc_fdw(4) <= fdw(11);
112  --slvo_sdlc_fdw(3) <= fdw(12);
113  --slvo_sdlc_fdw(2) <= fdw(13);
114  --slvo_sdlc_fdw(1) <= fdw(14);
115  --slvo_sdlc_fdw(0) <= fdw(15);
116  slvo_sdlc_fdw <= fdw;
117  both <= slvi_sdlc_dw & slvi_sdlc_wlk;
118
119  -- fdw logic
120  -- with busy signal
121  fdw <= dw_sum and ( not dw_product ) and ( not wlk_product ) and (
         not slvi_sdlc_wlk )
122          when sli_sdlc_busy = '0' else fdw;
123  -- without busy signal
124  --fdw <= dw_sum and ( not dw_product ) and ( not wlk_product ) and
         ( not slvi_sdlc_wlk );
125
126  c_edgetopulse : risingedge_to_pulse
127  port map (
128             sli_edge => sli_clk_div ,
129             sli_clk => sli_clk ,
130             sli_reset => sli_reset ,
131             slo_pulse => sample_start
132           );
```

```
133
134   --c_clkdivider : clkdivider port map (
135   --sli_clk => sli_clk ,
136   --sli_clk_div => sli_clk_div ,
137   --sli_reset => sli_reset ,
138   ----divide_val => X"0026259D" -- use 0x0026259D for 50ms period
139   --divide_val =>   X"0007CE0C" -- divider for 10.23MHz internal OSC
140   --);
141
142   -- one shift reg for each wlk and dw channel (32 total)
143   c_fdwcheck0 : fdwcheck
144   port map (
145             sli_clk => sli_clk ,
146             sli_reset => sli_reset ,
147             d_in => both(0) ,
148             sr_en => sr_en ,
149             sum => wlk_sum(0) ,
150             product => wlk_product(0)
151           );
152
153   c_fdwcheck1 : fdwcheck
154   port map (
155             sli_clk => sli_clk ,
156             sli_reset => sli_reset ,
157             d_in => both(1) ,
158             sr_en => sr_en ,
159             sum => wlk_sum(1) ,
160             product => wlk_product(1)
161           );
162
163   c_fdwcheck2 : fdwcheck
164   port map (
165             sli_clk => sli_clk ,
166             sli_reset => sli_reset ,
167             d_in => both(2)    ,
168             sr_en => sr_en ,
169             sum => wlk_sum(2) ,
170             product => wlk_product(2)
171           );
172
173   c_fdwcheck3 : fdwcheck
174   port map (
175             sli_clk => sli_clk ,
176             sli_reset => sli_reset ,
177             d_in => both(3)    ,
178             sr_en => sr_en ,
```

```
179             sum => wlk_sum(3) ,
180             product => wlk_product(3)
181           );
182
183   c_fdwcheck4 : fdwcheck
184   port map (
185             sli_clk => sli_clk ,
186             sli_reset => sli_reset ,
187             d_in => both(4) ,
188             sr_en => sr_en ,
189             sum => wlk_sum(4) ,
190             product => wlk_product(4)
191           );
192
193   c_fdwcheck5 : fdwcheck
194   port map (
195             sli_clk => sli_clk ,
196             sli_reset => sli_reset ,
197             d_in => both(5)    ,
198             sr_en => sr_en ,
199             sum => wlk_sum(5) ,
200             product => wlk_product(5)
201           );
202
203   c_fdwcheck6 : fdwcheck
204   port map (
205             sli_clk => sli_clk ,
206             sli_reset => sli_reset ,
207             d_in => both(6)    ,
208             sr_en => sr_en ,
209             sum => wlk_sum(6) ,
210             product => wlk_product(6)
211           );
212
213   c_fdwcheck7 : fdwcheck
214   port map (
215             sli_clk => sli_clk ,
216             sli_reset => sli_reset ,
217             d_in => both(7) ,
218             sr_en => sr_en ,
219             sum => wlk_sum(7) ,
220             product => wlk_product(7)
221           );
222
223   c_fdwcheck8 : fdwcheck
224   port map (
```

```
225             sli_clk => sli_clk ,
226             sli_reset => sli_reset ,
227             d_in => both(8)    ,
228             sr_en => sr_en ,
229             sum => wlk_sum(8) ,
230             product => wlk_product(8)
231           );
232
233   c_fdwcheck9 : fdwcheck
234   port map (
235             sli_clk => sli_clk ,
236             sli_reset => sli_reset ,
237             d_in => both(9)    ,
238             sr_en => sr_en ,
239             sum => wlk_sum(9) ,
240             product => wlk_product(9)
241           );
242
243   c_fdwcheck10 : fdwcheck
244   port map (
245             sli_clk => sli_clk ,
246             sli_reset => sli_reset ,
247             d_in => both(10) ,
248             sr_en => sr_en ,
249             sum => wlk_sum(10) ,
250             product => wlk_product(10)
251           );
252
253   c_fdwcheck11 : fdwcheck
254   port map (
255             sli_clk => sli_clk ,
256             sli_reset => sli_reset ,
257             d_in => both(11) ,
258             sr_en => sr_en ,
259             sum => wlk_sum(11) ,
260             product => wlk_product(11)
261           );
262
263   c_fdwcheck12 : fdwcheck
264   port map (
265             sli_clk => sli_clk ,
266             sli_reset => sli_reset ,
267             d_in => both(12) ,
268             sr_en => sr_en ,
269             sum => wlk_sum(12) ,
270             product => wlk_product(12)
```

```
271             );
272
273     c_fdwcheck13 : fdwcheck
274     port map (
275                 sli_clk => sli_clk ,
276                 sli_reset => sli_reset ,
277                 d_in => both(13) ,
278                 sr_en => sr_en ,
279                 sum => wlk_sum(13) ,
280                 product => wlk_product(13)
281             );
282
283     c_fdwcheck14 : fdwcheck
284     port map (
285                 sli_clk => sli_clk ,
286                 sli_reset => sli_reset ,
287                 d_in => both(14) ,
288                 sr_en => sr_en ,
289                 sum => wlk_sum(14) ,
290                 product => wlk_product(14)
291             );
292
293     c_fdwcheck15 : fdwcheck
294
295     port map (
296                 sli_clk => sli_clk ,
297                 sli_reset => sli_reset ,
298                 d_in => both(15) ,
299                 sr_en => sr_en ,
300                 sum => wlk_sum(15) ,
301                 product => wlk_product(15)
302             );
303
304     c_fdwcheck16 : fdwcheck
305     port map (
306                 sli_clk => sli_clk ,
307                 sli_reset => sli_reset ,
308                 d_in => both(16) ,
309                 sr_en => sr_en ,
310                 sum => dw_sum(0) ,
311                 product => dw_product(0)
312             );
313
314     c_fdwcheck17 : fdwcheck
315     port map (
316                 sli_clk => sli_clk ,
```

```
317             sli_reset => sli_reset ,
318             d_in => both(17) ,
319             sr_en => sr_en ,
320             sum => dw_sum(1) ,
321             product => dw_product(1)
322           );
323
324   c_fdwcheck18 : fdwcheck
325   port map (
326             sli_clk => sli_clk ,
327             sli_reset => sli_reset ,
328             d_in => both(18) ,
329             sr_en => sr_en ,
330             sum => dw_sum(2) ,
331             product => dw_product(2)
332           );
333
334   c_fdwcheck19 : fdwcheck
335   port map (
336             sli_clk => sli_clk ,
337             sli_reset => sli_reset ,
338             d_in => both(19) ,
339             sr_en => sr_en ,
340             sum => dw_sum(3) ,
341             product => dw_product(3)
342           );
343
344   c_fdwcheck20 : fdwcheck
345   port map (
346             sli_clk => sli_clk ,
347             sli_reset => sli_reset ,
348             d_in => both(20) ,
349             sr_en => sr_en ,
350             sum => dw_sum(4) ,
351             product => dw_product(4)
352           );
353
354   c_fdwcheck21 : fdwcheck
355   port map (
356             sli_clk => sli_clk ,
357             sli_reset => sli_reset ,
358             d_in => both(21) ,
359             sr_en => sr_en ,
360             sum => dw_sum(5) ,
361             product => dw_product(5)
362           );
```

```
363
364     c_fdwcheck22 : fdwcheck
365     port map (
366                 sli_clk => sli_clk ,
367                 sli_reset => sli_reset ,
368                 d_in => both(22) ,
369                 sr_en => sr_en ,
370                 sum => dw_sum(6) ,
371                 product => dw_product(6)
372               );
373
374     c_fdwcheck23 : fdwcheck
375     port map (
376                 sli_clk => sli_clk ,
377                 sli_reset => sli_reset ,
378                 d_in => both(23) ,
379                 sr_en => sr_en ,
380                 sum => dw_sum(7) ,
381                 product => dw_product(7)
382               );
383
384     c_fdwcheck24 : fdwcheck
385     port map (
386                 sli_clk => sli_clk ,
387                 sli_reset => sli_reset ,
388                 d_in => both(24) ,
389                 sr_en => sr_en ,
390                 sum => dw_sum(8) ,
391                 product => dw_product(8)
392               );
393
394     c_fdwcheck25 : fdwcheck
395     port map (
396                 sli_clk => sli_clk ,
397                 sli_reset => sli_reset ,
398                 d_in => both(25) ,
399                 sr_en => sr_en ,
400                 sum => dw_sum(9) ,
401                 product => dw_product(9)
402               );
403
404     c_fdwcheck26 : fdwcheck
405     port map (
406                 sli_clk => sli_clk ,
407                 sli_reset => sli_reset ,
408                 d_in => both(26) ,
```

```
409             sr_en => sr_en ,
410             sum => dw_sum(10) ,
411             product => dw_product(10)
412           );
413
414   c_fdwcheck27 : fdwcheck
415   port map (
416             sli_clk => sli_clk ,
417             sli_reset => sli_reset ,
418             d_in => both(27) ,
419             sr_en => sr_en ,
420             sum => dw_sum(11) ,
421             product => dw_product(11)
422           );
423
424   c_fdwcheck28 : fdwcheck
425   port map (
426             sli_clk => sli_clk ,
427             sli_reset => sli_reset ,
428             d_in => both(28) ,
429             sr_en => sr_en ,
430             sum => dw_sum(12) ,
431             product => dw_product(12)
432           );
433   c_fdwcheck29 : fdwcheck
434   port map (
435             sli_clk => sli_clk ,
436             sli_reset => sli_reset ,
437             d_in => both(29) ,
438             sr_en => sr_en ,
439             sum => dw_sum(13) ,
440             product => dw_product(13)
441           );
442
443   c_fdwcheck30 : fdwcheck
444   port map (
445             sli_clk => sli_clk ,
446             sli_reset => sli_reset ,
447             d_in => both(30) ,
448             sr_en => sr_en ,
449             sum => dw_sum(14) ,
450             product => dw_product(14)
451           );
452
453   c_fdwcheck31 : fdwcheck
454   port map (
```

```
455                 sli_clk => sli_clk ,
456                 sli_reset => sli_reset ,
457                 d_in => both(31) ,
458                 sr_en => sr_en ,
459                 sum => dw_sum(15) ,
460                 product => dw_product(15)
461               );
462
463
464 end Behavioral;
```

Listing B.4: Wishbone master finite state machine.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity wbmaster_fsm is
6    port(
7          sli_clk : in std_logic;
8          sli_reset : in std_logic;
9          wbc_en : out std_logic;
10         wbc_done : in std_logic;
11         wbc_src : out std_logic_vector( 7 downto 0 );
12         wbc_dest : out std_logic_vector( 7 downto 0 );
13         r0 : in std_logic_vector( 7 downto 0 );
14         r1 : in std_logic_vector( 7 downto 0 );
15         r2 : out std_logic_vector( 7 downto 0 )
16       );
17 end wbmaster_fsm;
18
19 architecture Behavioral of wbmaster_fsm is
20
21   type state_type is ( SETUP0 , SETUP1 , SETUP2 , IDLE ,
22                 DO_READ , CHECK_BUSY , DISCARD0 , DISCARD1 ,
23                 CHECK_TRRDY , GET_DATA , CHECK_TRRDYW ,
24                 GET_WBADDR , INC_WBADDRW , INC_WBADDRR ,
25                 WAIT_BUSYW , WAIT_BUSYR );
26   signal fsmstate : state_type;
27
28   -- Wishbone Address register signals
29   signal slvr_wbaddr : std_logic_vector( 7 downto 0 );
30   signal slv_wbaddr_next : std_logic_vector( 7 downto 0 );
31
32   -- I2C Wishbone register addresses
33   constant I2C_CMDR : std_logic_vector( 7 downto 0 ) := X"41";
34   constant I2C_IRQEN : std_logic_vector( 7 downto 0 ) := X"49";
35   constant I2C_IRQ : std_logic_vector( 7 downto 0 ) := X"48";
36   constant I2C_SR : std_logic_vector( 7 downto 0 ) := X"45";
37   constant I2C_RXDR : std_logic_vector( 7 downto 0 ) := X"47";
38   constant I2C_TXDR : std_logic_vector( 7 downto 0 ) := X"44";
39
40   -- Book keeping register addresses
41   constant R0_ADDR : std_logic_vector( 7 downto 0 ) := X"FF";
42   constant R1_ADDR : std_logic_vector( 7 downto 0 ) := X"FE";
43   constant R2_ADDR : std_logic_vector( 7 downto 0 ) := X"FD";
```

```
44
45 begin
46
47 nsl : process( sli_clk ) begin
48   if(   rising_edge( sli_clk ) ) then
49     if( sli_reset = '1' ) then
50       slvr_wbaddr <= ( others => '0' );
51       fsmstate <= SETUP0;
52     else
53       slvr_wbaddr <= slv_wbaddr_next;
54       case fsmstate is
55         when SETUP0 =>
56           if( wbc_done = '1' ) then
57             fsmstate <= SETUP1;
58           else
59             fsmstate <= SETUP0;
60           end if;
61         when SETUP1 =>
62           if( wbc_done = '1' ) then
63             fsmstate <= CHECK_BUSY;
64           else
65             fsmstate <= SETUP1;
66           end if;
67         when CHECK_BUSY =>
68           if( r0(6) = '0' and wbc_done = '1' ) then
69             fsmstate <= DISCARD0;
70           else
71             fsmstate <= CHECK_BUSY;
72           end if;
73         when DISCARD0 =>  -- Discard what is in the I2C_RX reg
74           if( wbc_done = '1' ) then
75             fsmstate <= DISCARD1;
76           else
77             fsmstate <= DISCARD0;
78           end if;
79         when DISCARD1 => -- Discard what is in the I2C_RX reg
80           if( wbc_done = '1' ) then
81             fsmstate <= SETUP2;
82           else
83             fsmstate <= DISCARD1;
84           end if;
85         when SETUP2 =>
86           if( wbc_done = '1' ) then
87             fsmstate <= IDLE;
88           else
89             fsmstate <= SETUP2;
```

```
90            end if;
91          --when IDLE =>
92          --if( r0(6) = '1' and wbc_done = '1' ) then
93          --fsmstate <= CHECK_TRRDY;
94          --else
95          --fsmstate <= IDLE;
96          --end if;
97          when IDLE =>
98            if( r0(6) = '1' and r0(2) = '1' and wbc_done = '1' ) then
99              if( r0(4) = '1' ) then -- SRW bit
100               fsmstate <= DO_READ;
101             else
102               fsmstate <= GET_WBADDR;
103             end if;
104           else
105             fsmstate <= IDLE;
106           end if;
107         when CHECK_TRRDY =>
108           if( r0(6) = '0' and wbc_done = '1' ) then
109             fsmstate <= IDLE;
110           elsif( r0(6) = '1' and r0(2) = '1' and wbc_done = '1' )
                then
111             if( r0(4) = '1' ) then -- SRW bit
112               fsmstate <= DO_READ;
113             else
114               fsmstate <= GET_WBADDR;
115             end if;
116           else
117             fsmstate <= CHECK_TRRDY;
118           end if;
119         when DO_READ => -- slave transmitting
120           if( wbc_done = '1' ) then
121             fsmstate <= INC_WBADDRR;
122           else
123             fsmstate <= DO_READ;
124           end if;
125         when INC_WBADDRR =>
126           fsmstate <= WAIT_BUSYR;
127         when WAIT_BUSYR =>
128           if( r0(6) = '0' and wbc_done = '1' ) then
129             fsmstate <= IDLE;
130           elsif( r0(6) = '1' and r0(2) = '1' and wbc_done = '1' )
                then
131             fsmstate <= DO_READ;
132           else
133             fsmstate <= WAIT_BUSYR;
```

```
134             end if;
135           when GET_WBADDR => -- slave receiving
136             if( wbc_done = '1' ) then
137               fsmstate <= CHECK_TRRDYW;
138             else
139               fsmstate <= GET_WBADDR;
140             end if;
141           when CHECK_TRRDYW =>
142             if( r0(6) = '0' and wbc_done = '1' ) then
143               fsmstate <= IDLE;
144             elsif( r0(6) = '1' and r0(2) = '1' and wbc_done = '1' )
                  then
145               fsmstate <= GET_DATA;
146             else
147               fsmstate <= CHECK_TRRDYW;
148             end if;
149           when GET_DATA =>
150             if( wbc_done = '1' ) then
151               fsmstate <= WAIT_BUSYW;
152             else
153               fsmstate <= GET_DATA;
154             end if;
155           when WAIT_BUSYW =>
156             if( r0(6) = '0' and wbc_done = '1' ) then
157               fsmstate <= IDLE;
158             elsif( r0(6) = '1' and r0(2) = '1' and r0(4) = '0' and
                    wbc_done = '1' ) then
159               fsmstate <= INC_WBADDRW;
160             elsif( r0(6) = '1' and r0(2) = '1' and r0(4) = '1' and
                    wbc_done = '1' ) then
161               fsmstate <= DO_READ;
162             else
163               fsmstate <= WAIT_BUSYW;
164             end if;
165           when INC_WBADDRW =>
166             fsmstate <= GET_DATA;
167           when others =>
168             fsmstate <= IDLE;
169       end case;
170     end if;
171   end if;
172 end process;
173
174 slv_wbaddr_next <= r1 when fsmstate = GET_WBADDR else
175             ( slvr_wbaddr + "01" ) when (
176                   fsmstate = INC_WBADDRW
```

```
177                 or fsmstate = INC_WBADDRR )
178             else slvr_wbaddr;
179
180   ol : process( fsmstate , slvr_wbaddr ) begin
181     r2 <= ( others => '0' );
182     wbc_src <= ( others => '0' );
183     wbc_dest <= ( others => '0' );
184     wbc_en <= '0';
185     case fsmstate is
186       when SETUP0 =>
187         r2 <= X"04";
188         wbc_src <= R2_ADDR; --0xFD
189         wbc_dest <= I2C_CMDR; --0x41
190         wbc_en <= '1';
191       when SETUP1 =>
192         r2 <= X"00";
193         wbc_src <= R2_ADDR; -- 0xFD
194         wbc_dest <= I2C_IRQEN; -- 0x49
195         wbc_en <= '1';
196       when CHECK_BUSY =>
197         wbc_src <= I2C_SR; -- 0x45
198         wbc_dest <= R0_ADDR; --0xFF
199         wbc_en <= '1';
200       when DISCARD0 =>
201         wbc_src <= I2C_RXDR; -- 0x47
202         wbc_dest <= R1_ADDR; -- 0xFE
203         wbc_en <= '1';
204       when DISCARD1 =>
205         wbc_src <= I2C_RXDR; -- 0x47
206         wbc_dest <= R1_ADDR; -- 0xFE
207         wbc_en <= '1';
208       when SETUP2 =>
209         r2 <= X"04";
210         wbc_src <= R2_ADDR; -- 0xFD
211         wbc_dest <= I2C_CMDR; --0x41
212         wbc_en <= '1';
213       when IDLE =>
214         wbc_src <= I2C_SR;
215         wbc_dest <= R0_ADDR;
216         wbc_en <= '1';
217       when CHECK_TRRDY =>
218         wbc_src <= I2C_SR; --0x45
219         wbc_dest <= R0_ADDR; --0xff
220         wbc_en <= '1';
221       when GET_WBADDR =>
222         wbc_src <= I2C_RXDR;
```

```
223           wbc_dest <= R1_ADDR;
224           wbc_en <= '1';
225         when CHECK_TRRDYW =>
226           wbc_src <= I2C_SR;
227           wbc_dest <= R0_ADDR;
228           wbc_en <= '1';
229         when GET_DATA =>
230           wbc_src <= I2C_RXDR;
231           wbc_dest <= slvr_wbaddr;
232           wbc_en <= '1';
233         when WAIT_BUSYW =>
234           wbc_src <= I2C_SR;
235           wbc_dest <= R0_ADDR;
236           wbc_en <= '1';
237         when DO_READ =>
238           r2 <= X"12";
239           wbc_src <= slvr_wbaddr;
240           wbc_dest <= I2C_TXDR;
241           wbc_en <= '1';
242         when WAIT_BUSYR =>
243           wbc_src <= I2C_SR;
244           wbc_dest <= R0_ADDR;
245           wbc_en <= '1';
246         when others =>
247           r2 <= ( others => '0' );
248           wbc_src <= ( others => '0' );
249           wbc_dest <= ( others => '0' );
250           wbc_en <= '0';
251       end case;
252     end process;
253
254   end Behavioral;
```

Listing B.5: Wishbone controller.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity wbcontroller is
5    port(
6        -- WISHBONE controller signals
7          sli_wb_ack                    : in std_logic;
8          slci_wb_clk             : in std_logic;
9          sli_wb_rst                    : in std_logic;
10         slo_wb_cyc                    : out std_logic;
11         slo_wb_stb                    : out std_logic;
12         slo_wb_we                     : out std_logic;
13         slvo_wb_addr            : out std_logic_vector( 7 downto 0 );
14         slvi_wb_datain          : in  std_logic_vector( 7 downto 0 );
15         slvo_wb_dataout : out std_logic_vector( 7 downto 0 );
16
17       -- NON-WISHBONE Signals
18         sli_en                  : in std_logic;
19         slvi_addr_src   : in std_logic_vector( 7 downto 0 );
20         slvi_addr_dest  : in std_logic_vector( 7 downto 0 );
21         slo_done                      : out std_logic
22       );
23 end wbcontroller;
24
25 architecture Behavioral of wbcontroller is
26
27   type wbstate is ( IDLE , DELAY , READ_SLV , LATCH_SLV , WRITE_SLV ,
         PULSE_DONE );
28   signal fsmstate : wbstate;
29
30   signal slvr_data : std_logic_vector( 7 downto 0 );
31   signal slv_data_next : std_logic_vector( 7 downto 0 );
32 begin
33
34 nsl : process( slci_wb_clk ) begin
35   if( rising_edge( slci_wb_clk ) ) then
36     if( sli_wb_rst = '1' ) then
37       fsmstate <= IDLE;
38       slvr_data <= ( others => '0' );
39     else
40       slvr_data <= slv_data_next;
41       case fsmstate is
42         when IDLE =>
```

```
43            if( sli_en = '1' ) then
44              fsmstate <= READ_SLV;
45            else
46              fsmstate <= IDLE;
47            end if;
48          when READ_SLV =>
49            if( sli_wb_ack = '1' ) then
50              fsmstate <= DELAY;
51            else
52              fsmstate <= READ_SLV;
53            end if;
54          when DELAY =>
55            fsmstate <= LATCH_SLV;
56          when LATCH_SLV =>
57            fsmstate <= WRITE_SLV;
58          when WRITE_SLV =>          -- wait for ACK signal from slave
59            if( sli_wb_ack = '1' ) then
60              fsmstate <= PULSE_DONE;
61            else
62              fsmstate <= WRITE_SLV;
63            end if;
64          when PULSE_DONE =>
65            fsmstate <= IDLE;
66        end case;
67      end if; -- reset
68    end if; -- rising edge
69  end process;
70
71  slv_data_next <= slvi_wb_datain when fsmstate = DELAY else slvr_data;
72
73    ol : process( fsmstate , slvi_addr_dest , slvi_addr_src ) begin
74      slvo_wb_addr <= ( others => '0' );
75      slo_wb_cyc <= '0';
76      slo_wb_stb <= '0';
77      slo_wb_we <= '0';
78      slo_done <= '0';
79      case fsmstate is
80        when READ_SLV =>
81          slvo_wb_addr <= slvi_addr_src;
82          slo_wb_cyc <= '1';
83          slo_wb_stb <= '1';
84          slo_done <= '0';
85        when DELAY =>
86          slvo_wb_addr <= slvi_addr_src;
87          slo_wb_cyc <= '1';
88          slo_wb_stb <= '1';
```

```
 89          slo_done <= '0';
 90        when LATCH_SLV =>
 91          slvo_wb_addr <= slvi_addr_src;
 92          slo_done <= '0';
 93        when WRITE_SLV    =>
 94          slvo_wb_addr <= slvi_addr_dest;
 95          slo_wb_we <= '1';
 96          slo_wb_cyc <= '1';
 97          slo_wb_stb <= '1';
 98          slo_done <= '0';
 99        when PULSE_DONE =>
100          slvo_wb_addr <= slvi_addr_dest;
101          slo_wb_we <= '1';
102          slo_wb_cyc <= '1';
103          slo_wb_stb <= '1';
104          slo_done <= '1';
105        when others =>
106
107      end case;
108    end process;
109
110    slvo_wb_dataout <= slvr_data;
111  end Behavioral;
```

Listing B.6: Wishbone slave output register.

```
library ieee;
use ieee.std_logic_1164.all;

entity wbslave8_out is
  port (
        -- WISHBONE Signals
          sloz_wb_ack                  : out std_logic;
          slci_wb_clk          : in  std_logic;
          sli_wb_rst                   : in  std_logic;
          sli_wb_cyc                   : in  std_logic;
          sli_wb_stb                   : in  std_logic;
          sli_wb_we                    : in  std_logic;
          slvi_wb_addr         : in  std_logic_vector( 7 downto 0 );
          slvi_wb_datain       : in  std_logic_vector( 7 downto 0 );
          slvoz_wb_dataout     : out std_logic_vector( 7 downto 0 );

        -- NON-WISHBONE Signals
          slvo_data                    : out std_logic_vector( 7
             downto 0 );
          slvi_addr_set        : in std_logic_vector( 7 downto 0 )
        );
end entity wbslave8_out;

architecture Behavioral of wbslave8_out is
  signal slr8_q : std_logic_vector( 7 downto 0 );
  signal sl8_q_next : std_logic_vector( 7 downto 0 );
  signal sl_ack : std_logic;
begin

REG : process( slci_wb_clk ) begin
  -- Assume there is no ack

  if( rising_edge( slci_wb_clk ) ) then
    --sl_ack <= '0';
    if( sli_wb_rst = '1' ) then
      -- Clear the register
      slr8_q <= ( others => '0' );
    else
      slr8_q <= sl8_q_next;
    end if;
  end if;
end process;

```

```
43 sl8_q_next <= slvi_wb_datain when ( sli_wb_stb = '1'
44     and sli_wb_cyc = '1'
45     and sli_wb_we = '1'
46     and slvi_wb_addr = slvi_addr_set )
47   else
48     slr8_q;
49     sl_ack <= '1' when sli_wb_stb = '1' and sli_wb_cyc = '1' and
           slvi_wb_addr = slvi_addr_set
50               else '0';
51
52     -- Only assume control of these lines when being addressed
53     sloz_wb_ack <= sl_ack when slvi_wb_addr = slvi_addr_set else 'Z';
54     slvoz_wb_dataout <= slr8_q when slvi_wb_addr = slvi_addr_set else
           "ZZZZZZZZ";
55
56     -- Constantly output what is in the register for non-wishbone
           transactions
57     slvo_data <= slr8_q;
58
59 end Behavioral;
```

Listing B.7: Wishbone slave input register.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity wbslave8_in is
5    port (
6         -- WISHBONE Signals
7           sloz_wb_ack                    : out std_logic;
8         --slci_wb_clk             : in  std_logic;
9         --sli_wb_rst                    : in  std_logic;
10        --sli_wb_cyc                    : in  std_logic;
11        --sli_wb_stb                    : in  std_logic;
12        --sli_wb_we                     : in  std_logic;
13          slvi_wb_addr             : in  std_logic_vector( 7 downto 0 );
14        --slvi_wb_datain           : in  std_logic_vector( 7 downto 0 );
15          slvoz_wb_dataout         : out std_logic_vector( 7 downto 0 );
16
17        -- NON-WISHBONE Signals
18          slvi_data                      : in std_logic_vector( 7
              downto 0 );
19          slvi_addr_set            : in std_logic_vector( 7 downto 0 )
20        );
21 end entity wbslave8_in;
22
23 architecture Behavioral of wbslave8_in is
24 begin
25   -- Only assume control of these lines when being addressed. Input
       data is
26   -- always valid
27   sloz_wb_ack <= '1' when slvi_wb_addr = slvi_addr_set else 'Z';
28   slvoz_wb_dataout      <=      slvi_data       when slvi_wb_addr =
       slvi_addr_set
29                       else      "ZZZZZZZZ"      ;
30   end Behavioral;
```

Listing B.8: Conflict check module.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity conflictcheck is
6    port(
7          sli_clk : in std_logic;
8          sli_clk_div : in std_logic;
9          sli_rst : in std_logic;
10
11         slo_conflict : out std_logic;
12
13         slvi_dw : in std_logic_vector( 15 downto 0 );
14         slvi_w : in std_logic_vector( 15 downto 0 );
15
16         slvo_w : out std_logic_vector( 15 downto 0 );
17         slvo_dw : out std_logic_vector( 15 downto 0 )
18       );
19 end conflictcheck;
20
21 architecture behavior of conflictcheck is
22
23   component risingedge_to_pulse
24     port(
25           sli_edge : in std_logic;
26           sli_clk : in std_logic;
27           sli_reset : in std_logic;
28           slo_pulse : out std_logic
29         );
30   end component;
31
32   signal slv_concount : std_logic_vector( 1 downto 0 ); -- allow 3
       samples of conflict
33   signal sl_conflict : std_logic;
34   signal sl_sample : std_logic;
35
36   -- 16 bit register signals
37   signal slr16w_q : std_logic_vector( 15 downto 0 );
38   signal slr16w_q_next : std_logic_vector( 15 downto 0 );
39   signal slr16dw_q : std_logic_vector( 15 downto 0 );
40   signal slr16dw_q_next : std_logic_vector( 15 downto 0 );
41
42 begin
```

```
43
44    slvo_w <= slr16w_q;
45    slvo_dw <= slr16dw_q;
46
47    sl_conflict <= '1' when slvi_dw( 0 ) = '1' and slvi_w( 0 ) = '1'
          else
48                   '1' when slvi_dw( 1 ) = '1' and slvi_w( 1 ) = '1'
                        else
49                   '1' when slvi_dw( 2 ) = '1' and slvi_w( 2 ) = '1'
                        else
50                   '1' when slvi_dw( 3 ) = '1' and slvi_w( 3 ) = '1'
                        else
51                   '1' when slvi_dw( 4 ) = '1' and slvi_w( 4 ) = '1'
                        else
52                   '1' when slvi_dw( 5 ) = '1' and slvi_w( 5 ) = '1'
                        else
53                   '1' when slvi_dw( 6 ) = '1' and slvi_w( 6 ) = '1'
                        else
54                   '1' when slvi_dw( 7 ) = '1' and slvi_w( 7 ) = '1'
                        else
55                   '1' when slvi_dw( 8 ) = '1' and slvi_w( 8 ) = '1'
                        else
56                   '1' when slvi_dw( 9 ) = '1' and slvi_w( 9 ) = '1'
                        else
57                   '1' when slvi_dw( 10 ) = '1' and slvi_w( 10 ) = '1'
                        else
58                   '1' when slvi_dw( 11 ) = '1' and slvi_w( 11 ) = '1'
                        else
59                   '1' when slvi_dw( 12 ) = '1' and slvi_w( 12 ) = '1'
                        else
60                   '1' when slvi_dw( 13 ) = '1' and slvi_w( 13 ) = '1'
                        else
61                   '1' when slvi_dw( 14 ) = '1' and slvi_w( 14 ) = '1'
                        else
62                   '1' when slvi_dw( 15 ) = '1' and slvi_w( 15 ) = '1'
                        else
63                   '0';
64
65    c_edgetopulse : risingedge_to_pulse
66    port map (
67                sli_edge => sli_clk_div ,
68                sli_clk => sli_clk ,
69                sli_reset => sli_rst ,
70                slo_pulse => sl_sample
71              );
72
```

```
73    -- register the outputs and do not change if a conflict is
          dettected within 400ms
74    p_conflict : process( sli_clk , sl_conflict , slv_concount )
75    begin
76      if( rising_edge( sli_clk ) ) then
77        if( sli_rst = '1' ) then
78          slv_concount <= "00";
79          slo_conflict <= '1';
80          slr16w_q <= ( others => '0' );
81          slr16dw_q <= ( others => '0' );
82        else
83          if( sl_conflict = '1' and slv_concount < "11" ) then
84            slo_conflict <= '1';
85            if( sl_sample = '1' ) then
86              slv_concount <= slv_concount + "01";
87            end if;
88          else
89            slv_concount <= "00";
90            slo_conflict <= '0';
91
92          end if;
93          slr16w_q <= slr16w_q_next;
94          slr16dw_q <= slr16dw_q_next;
95        end if; -- reset
96      end if; -- rising edge
97    end process;
98
99    slr16w_q_next <= slvi_w when sl_conflict = '0' else slr16w_q;
100   slr16dw_q_next <= slvi_dw when sl_conflict = '0' else slr16dw_q;
101
102
103 end behavior;
```

Listing B.9: Module sets a bit after 8 bits are passed to it.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity count8 is
6    port (
7           sli_tocount : in std_logic;
8           slo_count8 : out std_logic;    -- set after 8 bits are
                passed
9           sli_reset : in std_logic;
10          sli_clk : in std_logic
11        );
12 end count8;
13
14 architecture Behavioral of count8 is
15   signal slv_reg : unsigned( 3 downto 0 );
16   signal slv_reg_next : unsigned( 3 downto 0 );
17 begin
18   process( sli_clk )
19   begin
20     if( rising_edge( sli_clk ) ) then
21       if( sli_reset = '1' ) then
22         slv_reg <= ( others => '0' );
23       elsif( sli_tocount = '1' ) then
24         slv_reg <= slv_reg_next;
25       end if;
26     end if;
27   end process;
28   slv_reg_next <= slv_reg + 1;
29   slo_count8 <= '1' when std_logic_vector( slv_reg ) = "1000" else
       '0';
30 end Behavioral;
```

Listing B.10: Module converts a rising edge to a single pulse.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity risingedge_to_pulse is
5    port (
6           sli_edge : in std_logic;
7
8           slo_pulse : out std_logic;
9
10          sli_clk : in std_logic;
11          sli_reset : in std_logic
12        );
13 end risingedge_to_pulse;
14
15 architecture Behavioral of risingedge_to_pulse is
16   signal slvr_clkreg : std_logic_vector( 1 downto 0 );
17 begin
18   p_2bitsreg : process( sli_clk )
19   begin
20     if( rising_edge( sli_clk ) ) then
21       if( sli_reset = '1' ) then
22         slvr_clkreg <= ( others => '0' );
23       else
24         slvr_clkreg <= sli_edge & slvr_clkreg( 1 );
25       end if;
26     end if;
27   end process;
28
29   slo_pulse <= slvr_clkreg(1) and not slvr_clkreg(0);
30
31 end Behavioral;
```

Listing B.11: 8-bit serial in parallel out register.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity siporeg8 is
5    port (
6           sli_din : in std_logic;
7
8           slvo_data : out std_logic_vector( 7 downto 0 );
9
10          sli_en : in std_logic;
11          sli_reset : in std_logic;
12          sli_clk : in std_logic
13        );
14 end siporeg8;
15
16 architecture Behavioral of siporeg8 is
17   signal slvr_data : std_logic_vector( 7 downto 0 );
18   signal slv_next : std_logic_vector( 7 downto 0 );
19 begin
20   p_reg : process( sli_clk )
21   begin
22     if( rising_edge( sli_clk ) ) then
23       if( sli_reset = '1' ) then
24         slvr_data <= ( others => '0' );
25       else
26         slvr_data <= slv_next;
27       end if;
28     end if;
29   end process;
30   slv_next <= slvr_data( 6 downto 0 ) & sli_din when sli_en = '1'
       else slvr_data;
31   slvo_data <= slvr_data;
32 end Behavioral;
```

Listing B.12: SDLC cyclic redundancy check module.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity sdlccrc is
5    port (
6           sli_sdlc_data : in std_logic;
7           sli_clock : in std_logic;
8           sli_crc_en : in std_logic;
9
10          sli_reset : in std_logic;
11          sli_clearcrc : in std_logic;
12
13          slo_crc_pass : out std_logic
14        );
15 end sdlccrc;
16
17 architecture Behavioral of sdlccrc is
18
19   -- CRC-16 CCITT (JWP)
20   signal slv_crc : std_logic_vector( 15 downto 0 );
21   signal sl_xor16 : std_logic;
22   signal sl_xor12 : std_logic;
23   signal sl_xor5 : std_logic;
24   signal sl_xor0 : std_logic;
25
26   signal slv_crc_msb : std_logic_vector( 15 downto 0 );
27   signal sl_crc_pass_en : std_logic;
28   signal slr_crc_pass : std_logic;
29   signal sl_crc_pass_next       : std_logic;
30
31 begin
32
33   slo_crc_pass <= slr_crc_pass;
34
35   slv_crc_msb( 0 ) <= slv_crc( 15 );
36   slv_crc_msb( 1 ) <= slv_crc( 14 );
37   slv_crc_msb( 2 ) <= slv_crc( 13 );
38   slv_crc_msb( 3 ) <= slv_crc( 12 );
39   slv_crc_msb( 4 ) <= slv_crc( 11 );
40   slv_crc_msb( 5 ) <= slv_crc( 10 );
41   slv_crc_msb( 6 ) <= slv_crc( 9 );
42   slv_crc_msb( 7 ) <= slv_crc( 8 );
43   slv_crc_msb( 8 ) <= slv_crc( 7 );
```

```
44    slv_crc_msb( 9 ) <= slv_crc( 6 );
45    slv_crc_msb( 10 ) <= slv_crc( 5 );
46    slv_crc_msb( 11 ) <= slv_crc( 4 );
47    slv_crc_msb( 12 ) <= slv_crc( 3 );
48    slv_crc_msb( 13 ) <= slv_crc( 2 );
49    slv_crc_msb( 14 ) <= slv_crc( 1 );
50    slv_crc_msb( 15 ) <= slv_crc( 0 );
51
52    -- CRC xor bits
53    sl_xor16 <= slv_crc(15) xor sli_sdlc_data;
54    sl_xor12 <= slv_crc(11) xor sl_xor16;
55    sl_xor5 <= slv_crc(4) xor sl_xor16;
56    sl_xor0 <= sl_xor16 xor '0';
57
58    -- CRC Shift Register
59    p_crc : process( sli_clock )
60    begin
61      if( rising_edge( sli_clock ) ) then
62        if( sli_reset = '1' or sli_clearcrc = '1' ) then
63          slv_crc <= (others => '1');
64        elsif( sli_crc_en = '1' ) then
65          slv_crc <= slv_crc(14 downto 12) & sl_xor12 & slv_crc(10
                downto 5)
66                    & sl_xor5 & slv_crc(3 downto 0) & sl_xor0;
67        end if;
68      end if;
69    end process;
70
71    -- Keep the crc_pass high until it is checked
72    p_crc_pass_reg : process( sli_clock )
73    begin
74      if( rising_edge( sli_clock ) ) then
75        if( sli_reset = '1' or sli_clearcrc = '1' ) then
76          slr_crc_pass <= '0';
77        else
78          slr_crc_pass <= sl_crc_pass_next;
79        end if;
80      end if;
81    end process;
82
83    sl_crc_pass_next <= '1' when sl_crc_pass_en = '1' else slr_crc_pass
        ;
84
85    sl_crc_pass_en <= '1' when slv_crc_msb = X"F0B8" else '0';
86
87  end Behavioral;
```

Listing B.13: SDLC 0 bit insertion handler module.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity destuffer is
5    port (
6           sli_data : in std_logic;
7           sli_clockpulse : in std_logic;
8           sli_en : in std_logic;
9
10          slo_data : out std_logic;
11          slo_clockpulse : out std_logic;
12          slo_bse : out std_logic;
13
14          sli_clk : in std_logic;
15          sli_reset : in std_logic
16        );
17 end destuffer;
18
19 architecture Behavioral of destuffer is
20   signal slvr_bits : std_logic_vector( 5 downto 0 );
21   signal slv_nextbits : std_logic_vector( 5 downto 0 );
22 begin
23   slo_data <= sli_data;
24
25   p_bitsreg : process( sli_clk )
26   begin
27     if( rising_edge( sli_clk ) ) then
28       if( sli_reset = '1' or sli_en = '0' ) then
29         slvr_bits <= ( others => '0' );
30       else
31         slvr_bits <= slv_nextbits;
32       end if;
33     end if;
34   end process;
35
36   slv_nextbits <= sli_data & slvr_bits( 5 downto 1 )
37                   when (sli_clockpulse = '1' and sli_en = '1')
38                 else slvr_bits;
39
40     slo_clockpulse <= '0' when ( slv_nextbits = "011111" or sli_en =
         '0' ) else sli_clockpulse;
41
42     slo_bse <= '1' when slv_nextbits = "111111" else '0';
```

```
43
44      end Behavioral;
```

Listing B.14: SDLC packet flag detector module.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity flagdetector is
5    port (
6           sli_data : in std_logic;
7           sli_clockpulse : in std_logic;
8
9           slo_flag : out std_logic;
10
11          sli_reset : in std_logic;
12          sli_clk : in std_logic
13        );
14 end flagdetector;
15
16 architecture Behavioral of flagdetector is
17   -- Contents of the sipo register
18   signal slv_data : std_logic_vector( 7 downto 0 );
19   -- '1' when a flag is in the data register
20   signal sl_flag : std_logic;
21
22   -- Flag to watch for.
23   constant STARTSTOPFLAG          : std_logic_vector( 7 downto 0 ) := "
       01111110";
24
25   -- Serial in parallel out register to watch.
26   component siporeg8
27     port(
28           sli_din : in std_logic;
29
30           slvo_data : out std_logic_vector( 7 downto 0 );
31
32           sli_en : in std_logic;
33           sli_reset : in std_logic;
34           sli_clk : in std_logic
35         );
36   end component;
37   -- Convert level output to a pulse
38   component risingedge_to_pulse
39     port(
40           sli_edge : in std_logic;
41           sli_clk : in std_logic;
42           sli_reset : in std_logic;
```

```
43            slo_pulse : out std_logic
44          );
45    end component;
46  begin
47
48    sl_flag <= '1' when slv_data = STARTSTOPFLAG else '0';
49
50    c_data : siporeg8
51    port map (
52                sli_clk => sli_clk ,
53                sli_din => sli_data ,
54                sli_en => sli_clockpulse ,
55                sli_reset => sli_reset ,
56                slvo_data => slv_data
57              );
58
59    c_pulseout : risingedge_to_pulse
60    port map (
61                sli_edge => sl_flag ,
62                sli_clk => sli_clk ,
63                sli_reset => sli_reset ,
64                slo_pulse => slo_flag
65              );
66
67  end Behavioral;
```

Listing B.15: SDLC Type 129 message packet buffer.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity packetbuffer is
5    port (
6           slvi_data : in std_logic_vector( 7 downto 0 );
7           sli_push : in std_logic;
8
9           slvo_address : out std_logic_vector( 7 downto 0 );
10          slvo_control : out std_logic_vector( 7 downto 0 );
11          slvo_frame : out std_logic_vector( 7 downto 0 );
12          slvo_boi : out std_logic_vector( 31 downto 0 );
13
14          sli_reset : in std_logic;
15          sli_clk : in std_logic
16        );
17 end packetbuffer;
18
19 architecture Behavioral of packetbuffer is
20   signal slv_d0 : std_logic_vector( 7 downto 0 );
21   signal slv_d1 : std_logic_vector( 7 downto 0 );
22   signal slv_d2 : std_logic_vector( 7 downto 0 );
23   signal slv_d3 : std_logic_vector( 7 downto 0 );
24   signal slv_d4 : std_logic_vector( 7 downto 0 );
25   signal slv_d5 : std_logic_vector( 7 downto 0 );
26   signal slv_d6 : std_logic_vector( 7 downto 0 );
27   signal slv_d7 : std_logic_vector( 7 downto 0 );
28   signal slv_d8 : std_logic_vector( 7 downto 0 );
29   signal slv_d9 : std_logic_vector( 7 downto 0 );
30   signal slv_d10 : std_logic_vector( 7 downto 0 );
31   signal slv_d11 : std_logic_vector( 7 downto 0 );
32   signal slv_d12 : std_logic_vector( 7 downto 0 );
33   signal slv_d13 : std_logic_vector( 7 downto 0 );
34   signal slv_d14 : std_logic_vector( 7 downto 0 );
35
36   signal slvr_ring : std_logic_vector( 0 to 14 );
37   signal slv_ring : std_logic_vector( 0 to 14 );
38
39   component piporeg8
40     port (
41             slvi_data : in std_logic_vector( 7 downto 0 );
42             slvo_data : out std_logic_vector( 7 downto 0 );
43             sli_en : in std_logic;
```

```
44              sli_reset : in std_logic;
45              sli_clk : in std_logic
46            );
47    end component;
48
49  begin
50
51    slvo_address <= slv_d0;
52    slvo_control <= slv_d1;
53    slvo_frame <= slv_d2;
54    slvo_boi <= slv_d3 & slv_d4 & slv_d7 & slv_d8;
55
56    p_ring : process( sli_clk )
57    begin
58      if( rising_edge( sli_clk ) ) then
59        if( sli_reset = '1' ) then
60          slvr_ring <= ( 0=>'1', others => '0' );
61        elsif( sli_push = '1' ) then
62          slvr_ring <= '0' & slvr_ring( 0 to 13 );
63        end if;
64      end if;
65    end process;
66
67    slv_ring <= slvr_ring when sli_push = '1' else ( others => '0' );
68
69    c_d0 : piporeg8
70    port map (
71               slvi_data => slvi_data ,
72               slvo_data => slv_d0 ,
73               sli_en => slv_ring( 0 ) ,
74               sli_reset => sli_reset ,
75               sli_clk => sli_clk
76             );
77    c_d1 : piporeg8
78    port map (
79               slvi_data => slvi_data ,
80               slvo_data => slv_d1 ,
81               sli_en => slv_ring( 1 ) ,
82               sli_reset => sli_reset ,
83               sli_clk => sli_clk
84             );
85    c_d2 : piporeg8
86    port map (
87               slvi_data => slvi_data ,
88               slvo_data => slv_d2 ,
89               sli_en => slv_ring( 2 ) ,
```

```
90              sli_reset => sli_reset ,
91              sli_clk => sli_clk
92            );
93    c_d3 : piporeg8
94    port map (
95              slvi_data => slvi_data ,
96              slvo_data => slv_d3 ,
97              sli_en => slv_ring( 3 ) ,
98              sli_reset => sli_reset ,
99              sli_clk => sli_clk
100           );
101   c_d4 : piporeg8
102   port map (
103             slvi_data => slvi_data ,
104             slvo_data => slv_d4 ,
105             sli_en => slv_ring( 4 ) ,
106             sli_reset => sli_reset ,
107             sli_clk => sli_clk
108           );
109   c_d5 : piporeg8
110   port map (
111             slvi_data => slvi_data ,
112             slvo_data => slv_d5 ,
113             sli_en => slv_ring( 5 ) ,
114             sli_reset => sli_reset ,
115             sli_clk => sli_clk
116           );
117   c_d6 : piporeg8
118   port map (
119             slvi_data => slvi_data ,
120             slvo_data => slv_d6 ,
121             sli_en => slv_ring( 6 ) ,
122             sli_reset => sli_reset ,
123             sli_clk => sli_clk
124           );
125   c_d7 : piporeg8
126   port map (
127             slvi_data => slvi_data ,
128             slvo_data => slv_d7 ,
129             sli_en => slv_ring( 7 ) ,
130             sli_reset => sli_reset ,
131             sli_clk => sli_clk
132           );
133   c_d8 : piporeg8
134   port map (
135             slvi_data => slvi_data ,
```

```
136             slvo_data => slv_d8 ,
137             sli_en => slv_ring( 8 ) ,
138             sli_reset => sli_reset ,
139             sli_clk => sli_clk
140           );
141   c_d9 : piporeg8
142   port map (
143             slvi_data => slvi_data ,
144             slvo_data => slv_d9 ,
145             sli_en => slv_ring( 9 ) ,
146             sli_reset => sli_reset ,
147             sli_clk => sli_clk
148           );
149   c_d10 : piporeg8
150   port map (
151             slvi_data => slvi_data ,
152             slvo_data => slv_d10 ,
153             sli_en => slv_ring( 10 ) ,
154             sli_reset => sli_reset ,
155             sli_clk => sli_clk
156           );
157   c_d11 : piporeg8
158   port map (
159             slvi_data => slvi_data ,
160             slvo_data => slv_d11 ,
161             sli_en => slv_ring( 11 ) ,
162             sli_reset => sli_reset ,
163             sli_clk => sli_clk
164           );
165   c_d12 : piporeg8
166   port map (
167             slvi_data => slvi_data ,
168             slvo_data => slv_d12 ,
169             sli_en => slv_ring( 12 ) ,
170             sli_reset => sli_reset ,
171             sli_clk => sli_clk
172           );
173   c_d13 : piporeg8
174   port map (
175             slvi_data => slvi_data ,
176             slvo_data => slv_d13 ,
177             sli_en => slv_ring( 13 ) ,
178             sli_reset => sli_reset ,
179             sli_clk => sli_clk
180           );
181   c_d14 : piporeg8
```

```
182    port map (
183                slvi_data => slvi_data ,
184                slvo_data => slv_d14 ,
185                sli_en => slv_ring( 14 ) ,
186                sli_reset => sli_reset ,
187                sli_clk => sli_clk
188              );
189
190
191  end Behavioral;
```

Listing B.16: SDLC packet buffer.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity packetcheck is
5    port (
6           slvi_poi_address : in std_logic_vector( 7 downto 0 );
7           slvi_poi_control : in std_logic_vector( 7 downto 0 );
8           slvi_poi_frame : in std_logic_vector( 7 downto 0 );
9           slo_poi : out std_logic
10        );
11 end packetcheck;
12
13 architecture Behavioral of packetcheck is
14   --constant SDLCADDRESS : std_logic_vector( 7 downto 0 ) :=
       "00001000";
15   --constant SDLCCONTROL : std_logic_vector( 7 downto 0 ) :=
       "11000001";
16   --constant SDLCFRAME   : std_logic_vector( 7 downto 0 ) :=
       "10000001";
17   constant SDLCADDRESS : std_logic_vector( 0 to 7 ) := "00001000";
18   constant SDLCCONTROL : std_logic_vector( 0 to 7 ) := "11000001";
19   constant SDLCFRAME   : std_logic_vector( 0 to 7 ) := "10000001";
20 begin
21   slo_poi <= '1' when ( slvi_poi_address = SDLCADDRESS and
22               slvi_poi_control = SDLCCONTROL and
23               slvi_poi_frame = SDLCFRAME )
24             else '0';
25   end Behavioral;
```

Listing B.17: Module checks for SDLC Type 129 response message.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity packetcheck is
5    port (
6           slvi_poi_address : in std_logic_vector( 7 downto 0 );
7           slvi_poi_control : in std_logic_vector( 7 downto 0 );
8           slvi_poi_frame : in std_logic_vector( 7 downto 0 );
9           slo_poi : out std_logic
10        );
11 end packetcheck;
12
13 architecture Behavioral of packetcheck is
14   --constant SDLCADDRESS : std_logic_vector( 7 downto 0 ) :=
       "00001000";
15   --constant SDLCCONTROL : std_logic_vector( 7 downto 0 ) :=
       "11000001";
16   --constant SDLCFRAME   : std_logic_vector( 7 downto 0 ) :=
       "10000001";
17   constant SDLCADDRESS : std_logic_vector( 0 to 7 ) := "00001000";
18   constant SDLCCONTROL : std_logic_vector( 0 to 7 ) := "11000001";
19   constant SDLCFRAME   : std_logic_vector( 0 to 7 ) := "10000001";
20 begin
21   slo_poi <= '1' when ( slvi_poi_address = SDLCADDRESS and
22               slvi_poi_control = SDLCCONTROL and
23               slvi_poi_frame = SDLCFRAME )
24             else '0';
25   end Behavioral;
```

Listing B.18: 32-bit parallel in parallel out register.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity piporeg32 is
5    port (
6           slvi_data : in std_logic_vector( 31 downto 0 );
7
8           slvo_data : out std_logic_vector( 31 downto 0 );
9
10          sli_en : in std_logic;
11          sli_reset : in std_logic;
12          sli_clk : in std_logic
13        );
14 end piporeg32;
15
16 architecture Behavioral of piporeg32 is
17   signal slvr_data : std_logic_vector( 31 downto 0 );
18   signal slv_next : std_logic_vector( 31 downto 0 );
19 begin
20   p_reg : process( sli_clk )
21   begin
22     if( rising_edge( sli_clk ) ) then
23       if( sli_reset = '1' ) then
24         slvr_data <= ( others => '0' );
25       else
26         slvr_data <= slv_next;
27       end if;
28     end if;
29   end process;
30
31   slv_next <= slvi_data when sli_en = '1' else slvr_data;
32   slvo_data <= slvr_data;
33
34 end Behavioral;
```

Listing B.19: SDLC decoder finite state machine module.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity sdlcfsm is
5    port (
6           sli_bse : in std_logic;
7           sli_8bits : in std_logic;
8           sli_flag : in std_logic;
9           sli_poi : in std_logic;
10          sli_crc_pass : in std_logic;
11
12          slo_dse : out std_logic;
13          slo_resetbitcount : out std_logic;
14          slo_clearpacket : out std_logic;
15          slo_saveboi : out std_logic;
16          slo_push : out std_logic;
17
18          sli_reset : in std_logic;
19          sli_clk : in std_logic
20        );
21 end sdlcfsm;
22
23 architecture Behavioral of sdlcfsm is
24   type sdlcstate is ( IDLE , IN_PACKET , SAVE_BYTE , SAVE_BOI ,
       CHECKEOP );
25   signal fsmstate : sdlcstate;
26   signal fsmnextstate : sdlcstate;
27 begin
28
29   p_fsm : process( sli_clk )
30   begin
31     if( rising_edge( sli_clk ) ) then
32       if( sli_reset = '1' ) then
33         fsmstate <= IDLE;
34       else
35         fsmstate <= fsmnextstate;
36       end if;
37     end if;
38   end process;
39
40   p_fsmnext : process( fsmstate , sli_bse , sli_8bits , sli_flag ,
       sli_poi )
41   begin
```

```
42      case fsmstate is
43        when IDLE =>
44          if( sli_flag = '1' ) then
45            fsmnextstate <= IN_PACKET;
46          else
47            fsmnextstate <= IDLE;
48          end if;
49        when IN_PACKET =>
50          if( sli_8bits = '1' ) then
51            fsmnextstate <= SAVE_BYTE;
52          elsif( sli_bse = '1' ) then
53            fsmnextstate <= CHECKEOP;
54          else
55            fsmnextstate <= IN_PACKET;
56          end if;
57        when SAVE_BYTE =>
58          if( sli_bse = '1' ) then
59            fsmnextstate <= IDLE;
60          else
61            fsmnextstate <= IN_PACKET;
62          end if;
63        when SAVE_BOI =>
64          fsmnextstate <= IDLE;
65        when CHECKEOP =>
66          if( sli_8bits = '1' ) then
67            fsmnextstate <= IDLE;
68          elsif( sli_flag = '1' ) then
69            if( sli_poi = '1' ) then
70              fsmnextstate <= SAVE_BOI;
71            else
72              fsmnextstate <= IDLE;
73            end if;
74          else
75            fsmnextstate <= CHECKEOP;
76          end if;
77      end case;
78    end process;
79
80    p_outputs : process( fsmstate , sli_crc_pass )
81    begin
82      slo_dse <= '0';
83      slo_resetbitcount <= '0';
84      slo_clearpacket <= '0';
85      slo_saveboi <= '0';
86      slo_push <= '0';
87      case fsmstate is
```

```
88          when IDLE =>
89            slo_resetbitcount <= '1';
90            slo_clearpacket <= '1';
91          when IN_PACKET =>
92            slo_dse <= '1';
93          when SAVE_BYTE =>
94            slo_push <= '1';
95            slo_resetbitcount <= '1';
96            slo_dse <= '1';     -- hold high
97          when SAVE_BOI =>
98            slo_saveboi <= sli_crc_pass;
99          when CHECKEOP =>
100           slo_dse <= '1';
101       end case;
102     end process;
103
104
105
106 end Behavioral;
```

Listing B.20: Flashing don't walk decoder shift register.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  --use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity fdwcheck is
6    port (
7           sli_clk : in std_logic;
8           sli_reset : in std_logic;
9           d_in : in std_logic;
10          sr_en : in std_logic;
11
12          sum : out std_logic;
13          product : out std_logic
14        );
15 end fdwcheck;
16
17 architecture Behavioral of fdwcheck is
18
19   signal r_reg : std_logic_vector( 9 downto 0 );
20   signal r_next : std_logic_vector( 9 downto 0 );
21
22 begin
23
24   p_sr : process( sli_clk )
25   begin
26     if( rising_edge( sli_clk ) ) then
27       if( sli_reset = '1' ) then
28         r_reg <= ( others => '0' );
29       else
30         r_reg <= r_next;
31       end if; -- reset
32     end if; -- rising edge
33   end process;
34
35   r_next <= d_in & r_reg( 9 downto 1 ) when sr_en = '1' else r_reg;
36
37   sum <= '0' when r_reg = "0000000000" else '1';
38   product <= '1' when r_reg = "1111111111" else '0';
39
40 end Behavioral;
```

Listing B.21: 8-bit parallel in parallel out register.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity piporeg8 is
5    port (
6           slvi_data : in std_logic_vector( 7 downto 0 );
7
8           slvo_data : out std_logic_vector( 7 downto 0 );
9
10          sli_en : in std_logic;
11          sli_reset : in std_logic;
12          sli_clk : in std_logic
13        );
14 end piporeg8;
15
16 architecture Behavioral of piporeg8 is
17   signal slvr_data : std_logic_vector( 7 downto 0 );
18   signal slv_next : std_logic_vector( 7 downto 0 );
19 begin
20   p_reg : process( sli_clk )
21   begin
22     if( rising_edge( sli_clk ) ) then
23       if( sli_reset = '1' ) then
24         slvr_data <= ( others => '0' );
25       else
26         slvr_data <= slv_next;
27       end if;
28     end if;
29   end process;
30
31   slv_next <= slvi_data when sli_en = '1' else slvr_data;
32   slvo_data <= slvr_data;
33
34 end Behavioral;
```