Neural Autonomous Robot Controller
Final Report

Jason Gunthorpe <jgg@ualberta.ca>
Darren O'Reilly <oreilly@ee.ualberta.ca>

April 5, 2000

Declaration of Original Content

The design elements of this project and report are entirely the original work of the authors except as follows:

1. Carry Save Adder, originally implemented by Dr. Elliott [8]
2. Ryan Lewis made significant contributions to the Abstract, the Neural Network description, the time line and some of the past reports.
3. Figure 1 was prepared by Jeff Woo of the ARVP [11]
4. Hardware attachments were designed courtesy of the Mechanical Enginerring shop and Jeff Woo [11]
5. Portions of the DISPLAYER program are due to Jason Gunthorpe's work on the ARVP project and are taken verbatim. [14]

Signatures

- Jason Gunthorpe
- Darren O'Reilly

Abstract

The Autonomous Vehicle Project's Bear Cub robot[9] has been recently built. A simple system is constructed that is capable of controlling this vehicle to follow a single white line on a black background. The control system for the robot is Neural Network based. It will be able to follow lines that turn smoothly; similar to conditions that are encountered in normal driving situations. The robot is also capable of dealing with some simple input variations, such as breaks in the line. The controller takes its input from a 16 photo-transistor/infrared-led paired sensor array pointed at the ground directly in front of the vehicle which is interfaced through an 8 bit analog to digital converter. The robot is able to move at a variable speed forwards and turn at any angle. Skid steering is used on the Bear Cub making it capable of moving at several speeds in the forward and reverse direction.

Two chips are provided, both implemented on the ALTERA FLEX10K240-4 FPGA, one to collect training data with a base station and the other to actually run the robot in an autonomous mode. Training of the neural network is done offline using a normal UNIX workstation. The weights from the trained neural network are hard coded into the autonomous chip's ROM.

Contents

List of Figures

1 Achievements

The project implements both the training and autonomous (run) modes for the design. The training mode makes the robot operate as a teleoperated vehicle that can return sensory information back to the base station and the base station can send back guidance commands. This control mechanism is sufficient to allow a human being to navigate a course without actually seeing the robot. Control is performed via a radio link and a laptop running custom control software. Included in the training module is the implementation of the ADC control module, the serial input and output as well as the PWM control and generation modules.

After the data has been collected by the base station the neural network training program[10] is run on the collected data to generate a set of weights for the autonomous mode. After the weights are decided they are coded into the ROM of the NNRUN module and the robot is able to navigate autonmously. The neural network training program has been implemented and tested on sample data collected by the ARVP project and is believed to work.

A wire wrapped interfacing board was created that contains the ADC chip and the RS232 voltage converter as well as the support circuitry for the infrared phototransistors and LEDs. An 2×8 array of phototransitors/LED pairs was also constructed and connected to the interfacing board. The entire interfacing board hardware has been completely tested using the VHDL testing programs described later in this report. The entire hardware system has been completely verified up to the robot itself. The phototransistor array has also been mounted to the base of the robot[11].

Measurements of the actual photo transistor output have been done to ensure they are within the required voltage range of 0 to 5 volts.

Nearly all of the VHDL has been completely test benched and tested in hardware to demonstrate its correct function. A few functions still remain to be confirmed in the autonomous module. In generating the VHDL and test benches the PYTHON scripting language was employed to generate complex VHDL structures without much manual coding. Overall there are 31 VHDL modules comprising 3161 lines of code, 10 PYTHON helpers and 3 C modules.

2 Overview

Two chips have been implemented, the first is intended for collecting training data and allows teleoperation of the vehical over a radio link, while the other performs completely autonoumous control of the vehical based on a feed forward neural network implemented in the chip. Each chip has an identical pin out, 1 button, serial input and output, direction and PWM for each motor and an ADC interface. These connections leave the UP1 board and are connected to a wire wrap board which contains the interfacing hardware, RS-232 serial interface, ADC and photo sensor array.

The sensory input is recived from a 2×8 array of phototransistor/LED pairs which are sensitive to a particular infraded wavelength. When mounted very close to the ground they are able to sense a contrast change on the surface, due to the highly reflective white line that the robot is to follow. The output of each individual phototransistor is sampled by the ADC to build an internal surface intensity map of the ground directly under the sensor.

The robot itself is a skid steered 4 wheeled vehicle powered by 4 separate motors. Control of these motors is achieved using pulse width modulation (PWM) with a direction bit which permits a very wide range of motion. The chips however only implement 5 separate motion states to simplify the design, see Table 2 for details.

3 Description of Operation

3.1 Theory of Operation

The control system will be designed so that the robot can follow a white line (laid out on a high contrast environment) by using a photo-sensor array and a neural network built with a FPGA. This system will be implemented on the ARVP's 'Bear Cub' mobile platform which is a moderate sized skid steered vehicle.

Skid steering allows the robot to turn about its centre axis or make gentler curves by varying the relative velocity of the two sides of the vehicle. In order to simplify the neural network the control system for the robot will only output 5 distinct vector directions and a stop 'direction'. These directions are explained in Table 2. The design of the 'Bear Cub' robot has a total of 4 independently controllable wheels with position feedback. For the purposes of this project the wheel motors on each side will be paired up and the feedback ignored. Figure shows a diagram of the 'Bear Cub' robot.

In order to visualise the course that the robot is set to follow, an array of infrared photo-transistors and photo emitters will be placed low to the ground on the front of the vehicle. This array will then be scanned by a 16 input ADC to get a

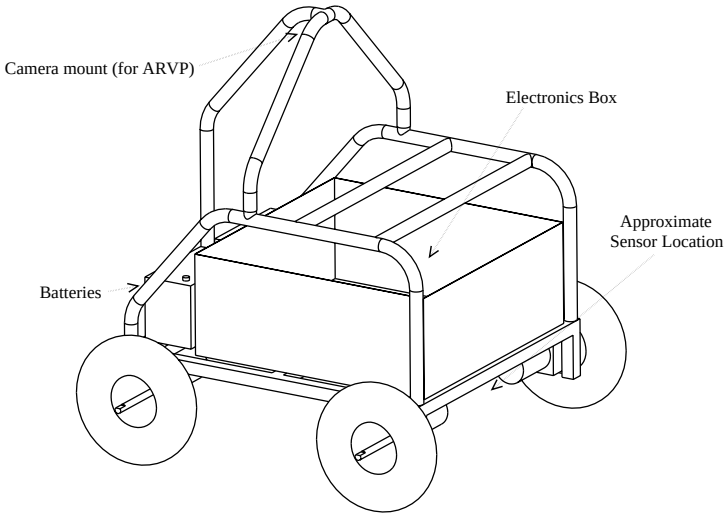


Figure 1: Diagram of the ARVP 'Bear Cub'[11]

signal from each point. Internally a matrix of points will be generated which corresponds roughly to 'how much white line' is in each cell, measured by intensity of reflected infrared light.

The optical sensors are configured in a photo-transistor/infrared led pairing. Each led sends out infrared light at the specific wavelength that the photo-transistors are sensitive to. The photo-transistors have a $\pm 12°$ of sensitivity to pick up the infrared light emitted by the LEDs, which have a $\pm 30°$ of transmission.

In order to train the neural network it is necessary to capture real data and motion control commands from a proper execution of a course. This is done with a special VHDL module that converts the platform into a remote controlled robot with sensory feedback to a base station over a radio link. The base station is responsible for capturing all of the images from the sensors and the motion commands given to the robot. Navigation is performed by having a human examine the output from the photo sensor array and make direction control decisions based on that. This helps to ensure the robot is trained based on what the system can see, not any sort of external observation.

3.2 Neural Network Design

The outputs of the neural network will determine the desired direction of robot movement. There are five possible directional movement choices: sharp left, slight left, go straight, slight right, and sharp right. These outputs will be translated into a directional vector which will determine the movement of the robot's wheels.

The neural network connections will be as shown in Figure 2. Note the notation of the Hidden Layer indicates a layer of neurons hidden from the outputs.[12] Note that the actual implementation of the network will have a variable number of neurons in the Hidden Layer, the specific selection will be done after training data is collected and analyzed.

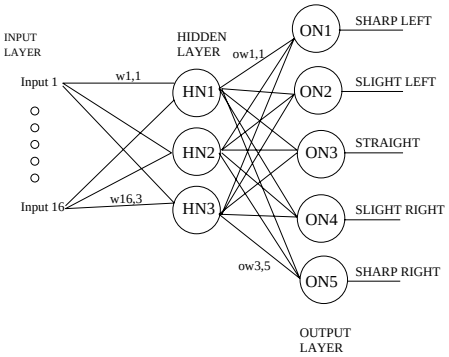


Figure 2: Neural Network Topology

The neural network controller will be initially trained with inputs and outputs that are collected from a human controlled run of the robot. The inputs will be determined by the data collected from the optical sensor array, and the desired outputs will be recorded based on the desired directional choice. The training for this network will be implemented using C, entirely separate from the FPGA coding. Based on the results of this training, the optimal neuron weights will be determined and coded into the FPGA. Ultimately, the robot will be able to interpret the input from the optical sensor array and translate this into the most appropriate output direction vector for navigation along a previously laid out white line.

The purpose of the FPGA is merely to act as a feed forward neural network processor. The training will employ a training style similar to what was used for the Neural Imaging Processor Project[2]. In the network, each neuron in the network will be equipped with the standard Sigmoid function (see Equation 1). Back propagation with standard gradient based learning will be used to train this network. Details of the back-propagation algorithm are online at Neural Network Tutorial for ARVP `http://ugweb.cs.ualberta.ca/~cbarton/arvp/tutorial.ps`. The use of this training algorithm will allow the computer to calculate the values for each weight of each connection in the network. These weights will be hard coded into the FPGA ROM based on the floating point weights from training. They will then be scaled appropriately and translated into 3-bit 2's complement numbers. Using these weights, the outputs of each neuron will be calculated in accordance with Equation 2, where the summation represents all of the connections to the given neuron. The Unit Step Function (see Equation 3) is used as the Neuron Function for the FPGA network. This is the same as the Sigmoid Function used in training (see Equation 1) with the slope set to infinity.

$$Sigmoid(x) = \frac{1}{1+exp(-x)} \tag{1}$$

$$O_{Neuron} = NeuronFunction(\sum_{i=1}^{n} Input_i \cdot w_i) \tag{2}$$

$$u(x) = \begin{cases} 1 & \text{for x} \geq 0 \\ 0 & \text{for x} < 0 \end{cases} \tag{3}$$

The Unit Step Function is used to remove any granularity of the neurons in the Neural Network, making it easy to implement digitally on the FPGA.

As above, there will be five outputs from the network: hard left, slight left, go straight, slight right, and hard right. Ideally, if the training of the network is successful, only one neuron will fire at a single instant of time, where the single firing neuron will dictate the desired direction of the robot. However, as a precaution, the robot is equipped with an output translation system that can account for multiple neuron firings. The translation consists of a truth table that will interpret the multiple fired neurons and translate it into a single direction vector as shown in Table 1.

3.3 Neural Processor Design

The implementation of the neural processor in VHDL is somewhat simplified. High performance hidden and output neurons are built using fully combinational logic. The input and output for this neuron are selected using a pair of multiplexors, while the weights are selected from onboard ROM. Once the ADC has finished sweeping the photo transistors the neural processor will iterate over all weights and store all the outputs. Processing of the output layer is done using the output version of the neuron, but otherwise using the same scheme.. Finally the output of this layer is passed through the translation table (see Table 1) and passed into the PWMCONTROL module to generate the proper PWM.

The most complicated part of this design is the 16 input multiply/add network. For speed and simplicity this is designed using carry-save arithmetic. The first part of the network takes in the 16 inputs and the 16 weights and generates 32 carry save outputs that represent the results of the 16 multiplications. These 32 outputs are then passed through a tree of 3 input, 2 output carry save adders to produce a final pair of outputs representing the neuron. This is then converted to a single binary output and truncated for the next stage. For the output neuron the multiply step is reduced to an and operation since the input is only a single bit.

Generation of both adder trees was done using a PYTHON script that creates the correct interconnections and components, there is a total of 30 carry save adders in the 32 input tree, each step reduces the number of inputs by $\frac{3}{2}$.

3.4 C Neural Network Software

The Neural Network C coding is split into three modules: *dignip-train.c*, *dignip-exec.c*, and *dignip-vhdl.c*. The code is designed for two purposes: one being a test version of a new network style for the Neural Image Processor (NIP) for the

Hard Left HL	Slight Left SL	Straight ST	Slight Right SL	Hard Right HR	Interpreted OUTPUT
0	0	0	0	0	ST
0	0	0	0	1	HR
0	0	0	1	0	SR
0	0	0	1	1	SR
0	0	1	0	0	ST
0	0	1	0	1	SR
0	0	1	1	0	SR
0	0	1	1	1	SR
0	1	0	0	0	SL
0	1	0	0	1	ST
0	1	0	1	0	ST
0	1	0	1	1	ST
0	1	1	0	0	SL
0	1	1	0	1	ST
0	1	1	1	0	ST
0	1	1	1	1	ST
1	0	0	0	0	HL
1	0	0	0	1	ST
1	0	0	1	0	ST
1	0	0	1	1	ST
1	0	1	0	0	SL
1	0	1	0	1	ST
1	0	1	1	0	ST
1	0	1	1	1	ST
1	1	0	0	0	SL
1	1	0	0	1	ST
1	1	0	1	0	ST
1	1	0	1	1	ST
1	1	1	0	0	SL
1	1	1	0	1	ST
1	1	1	1	0	ST
1	1	1	1	1	ST

Table 1: Neural Network translation table

Autonomous Robotic Vehicle Project, and the other is for this project. The majority of collected data is fed through the training module *dignip-train.c* which uses back propagation to train a digital style Neural Network to predict direction output for specific image input. Note that not all the data is fed through the training algorithm. This leaves several test points left over to check and see if the network has memorized or learned how to drive after training achieves convergence. It is important to distinguish that the network has learned and not memorized for this application to work - thus training data that the network has never seen is reserved for testing. After training, the weights are dumped to a text file which is read into *dignip-exec.c*. That program then takes an arbitrary test set and compares generated feed forward network output of each point to its direction tag. The first two modules utilize the floating point capabilites of a computer system which will not be available on the FPGA. After the weights have been verifed with the test set, they are converted to integers and the network is executed with *dignip-vhdl.c*. The output from this network is then verified again to check that the integer version generates acceptable output. From this, weights may have to be adjusted to work better in the VHDL implementation.

3.5 train Module

In order to train the Neural Network it is necessary to collect real life sensor data and motion commands from the robot. This is done using a special VHDL top level (chip) that allows teleoperation of the robot and collection of image data to a base station. This chip implements a pair of state machines, the first continuously collects data from the ADC and transmits it out the serial port, and the second takes motion control commands from the serial port and maps them to the proper PWM outputs on the wheels.

The overall structure of the entire training module, and all sub modules is shown in Figure 3. This figure shows the ports on each component and the overall structural representation of the TRAIN module, as well as the instantiation hierarchy

The data collection state machine (Figure 4) signals the ADC module to begin data collection, waits for collection to start, drops the begin signal and waits for completion. The data output is then clocked into the serial bit shifter and the state machine waits for the serial port to accept the data. Serial transmission is then done in parallel with the next ADC cycle.

The serial output from the ADC is a 7 bit number, the 8th bit is used by the ADC state machine to signal the base station that it is scanning cell 0. When the 0th cell is scanned the 7th bit is set low. This allows the base station to calculate which cell the ADC is currently scanning.

The motion control portion of the training module is mostly made of combinational logic. A rough diagram of its operation is shown in Figure 5. A byte is taken from the serial module, the lower 4 bits are taken and passed into a module that generates the correct PWM high time setting for that byte on each side of the robot. The raw byte is also sent to a hex LED interface for display. Each time a PWM cycle completes it takes the latest output from the key to high time converter (PWMCONTROL). The register connecting the serial decoder and the PWM generators is clocked only when new serial data is available.

These two processes operate continuously and in parallel, but a reset line is connected to them all from one of the push buttons on the UP1 board. The reset completely resets all modules. The base station is responsible for correctly sequencing motion events with vision data for later processing with the neural network training software.

3.6 nnrun Module

The top level module for autonomous control of the robot is the NNRUN module. It exports the same interface as the TRAIN module so the chip is compatible, but it does not use the serial ports or the LEDs. It integrates the entire NNET neural processor, output decoders and weight rom with a state machine that samples the ADC into a large register. Each computation cycle causes the ADC to sample all 16 inputs and store them. Then the neural processor operates on those inputs to yeild the 5 bit output which is then converted by the NNOUT module to a direction control. Figure 6 shows the overall structure of the NNRUN module and all sub modules.

3.7 ADC Module

The ADC (Analog to Digital Converter) module is responsible for interfacing with the 0817CCN[1] chip. It has a single output indicating that is ready to start a conversion. The handshaking process has the DO_CONVERSION signal asserted low, which causes READY to go low once conversion begins. After conversion is completed READY goes high and data is available on the 8 data lines from the ADC chip. Internally the ADC module generates a properly scaled down clock for the ADC chip and synchronises its control signal transitions with this clock. Figure 7 summarises the internal state transitions of the ADC module.

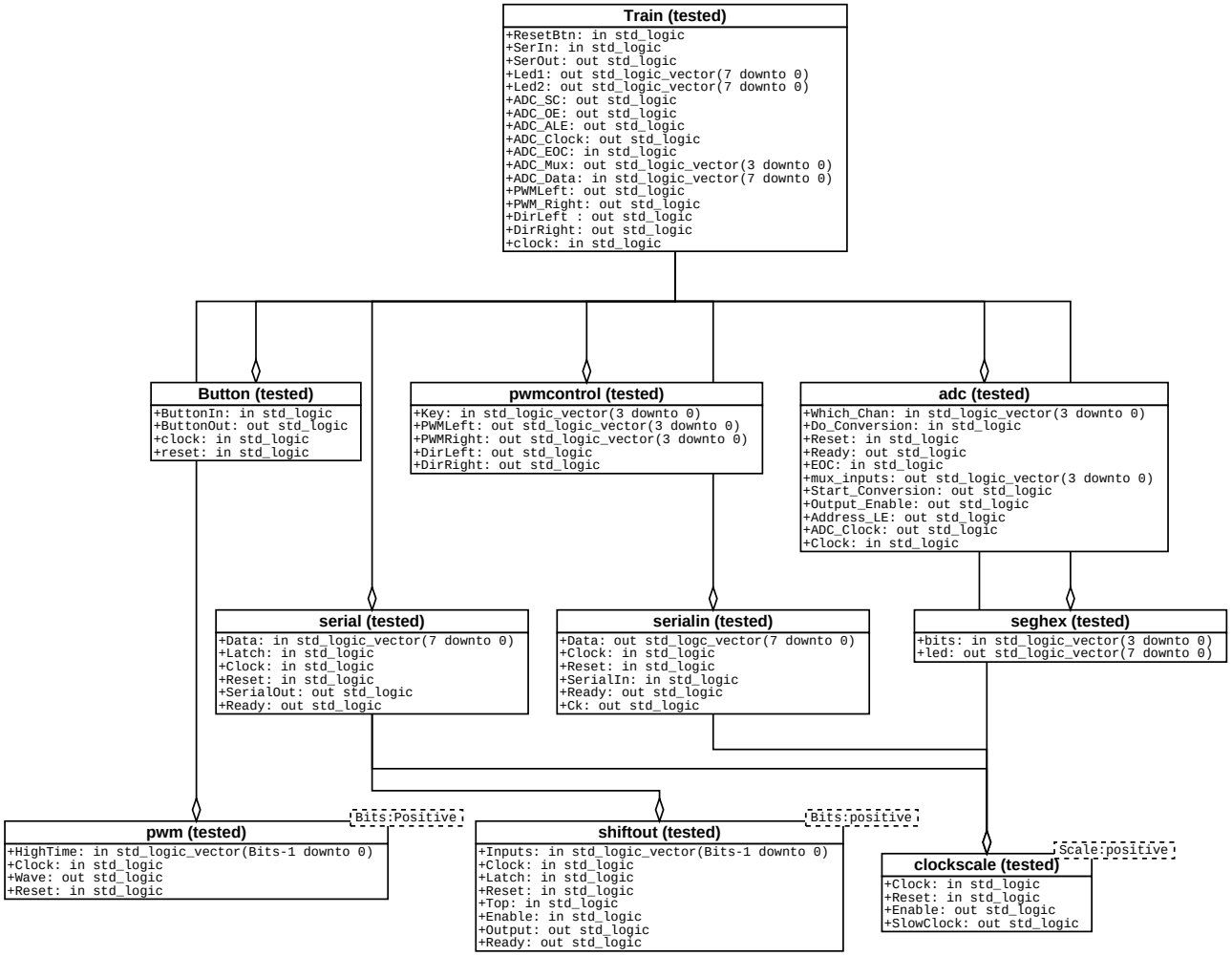


Figure 3: Training Module Structural Diagram

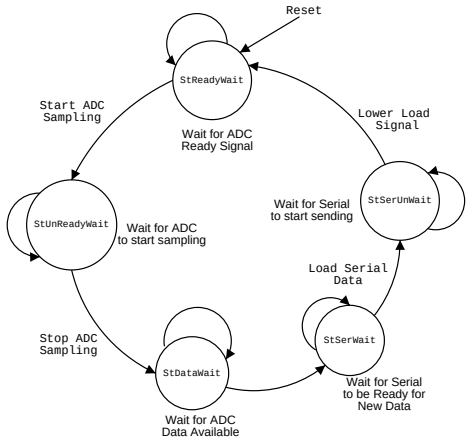


Figure 4: ADC Training State Machine

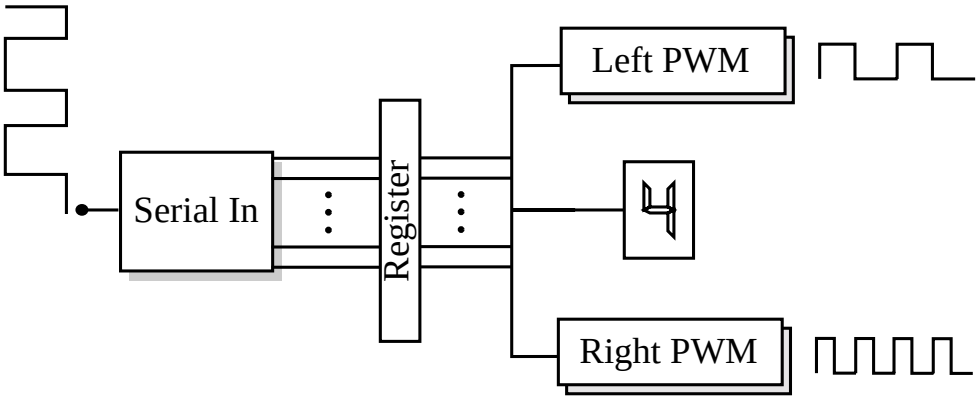


Figure 5: Training Moton Control

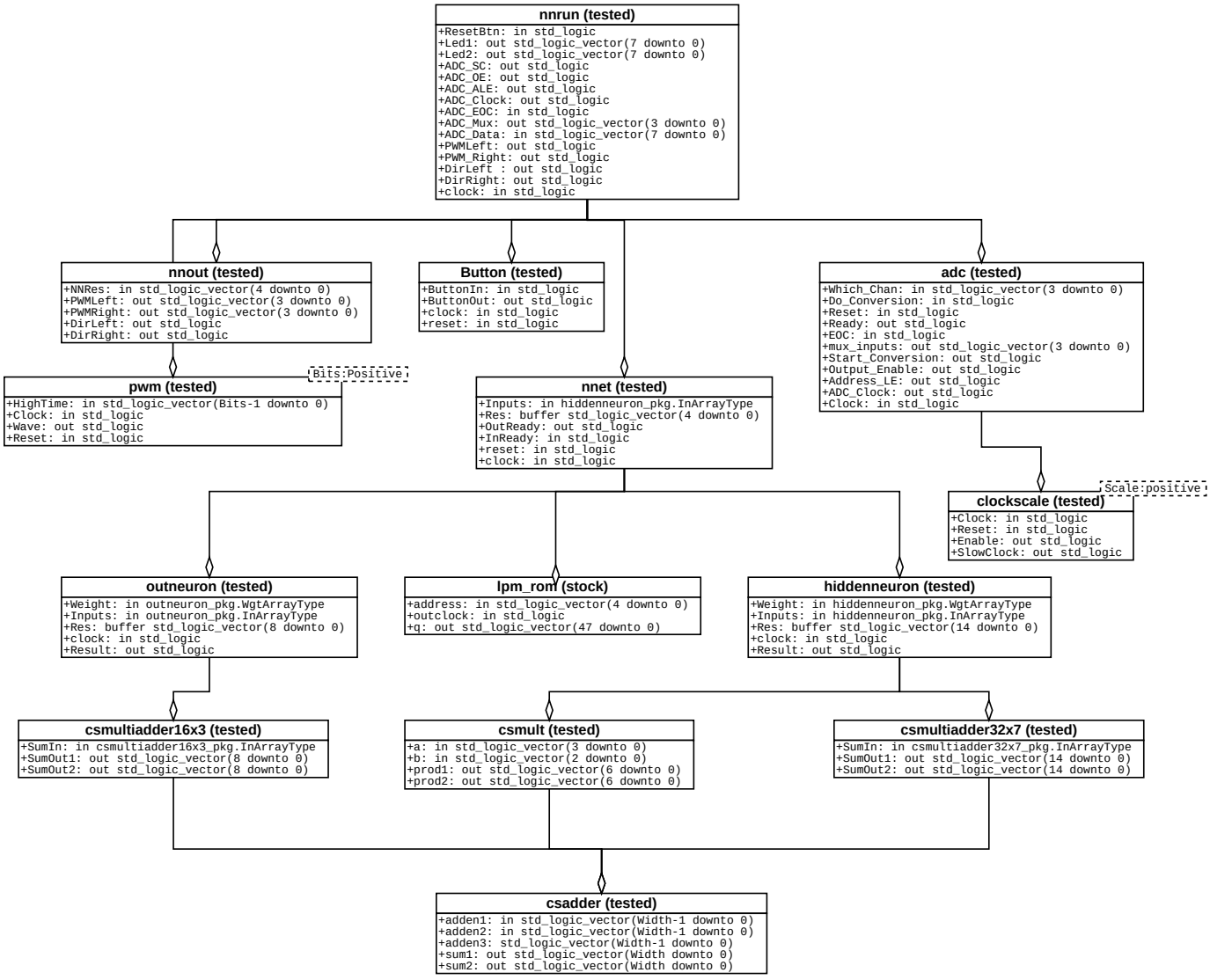


Figure 6: NNRun Structural Diagram

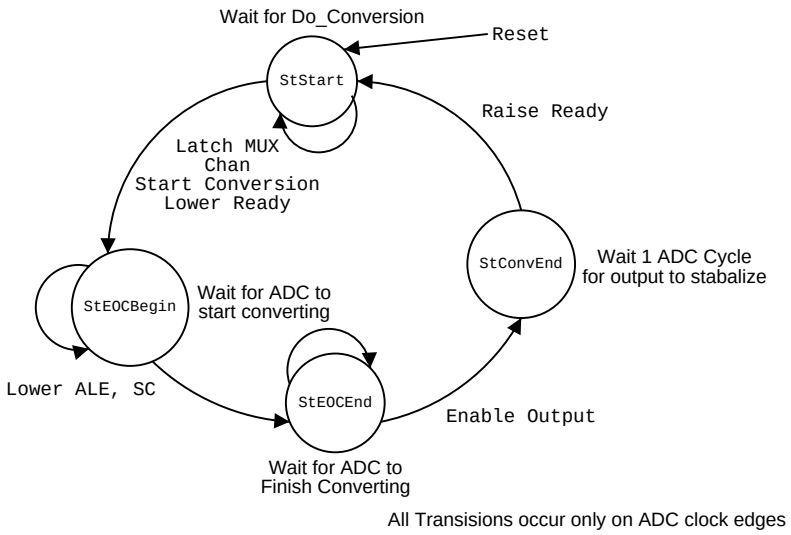


Figure 7: ADC State transitions

The remainder of the ADC control lines are directly connected to the chip, these are clocked according to the state diagram (Figure 7) and the chip data sheet.

3.8 button Module

This module is responsible for debouncing and un-inverting the button inputs from the push buttons. After the push button input is passed through this module it acts as a clean on/off value that is 1 while the push button is depressed and 0 otherwise. It operates by keeping an internal counter, the button input must remain stable for the set number of clock cycles before it is mapped to the module output. The BUTTON Module was created for EE 552 Lab 6[2].

3.9 pwmcontrol Module

The PWMCONTROL module, as mentioned above, takes a 4 bit value that corresponds to a motion direction and converts it into a pair of PWM high time values and a pair of direction bits. The conversion function is intended to logically map to the ASCII values 0-9 for use by a numeric keypad (for instance, Up is 8). Note that in the ASCII table the values 0-9 are ASCII values 0x30-0x39, thus the 4 bit truncation exactly matches the key that is pressed. Table 2 summaries this transformation.

Key	Name	Left Direction	Left Speed	Right Direction	Right Speed
5	Stop	-	0	-	0
4	Hard Left (HL)	Reverse	8	Foward	15
7	Slight Left (SL)	Forward	4	Forward	12
8	Forward (ST)	Forward	15	Forward	15
9	Slight Right (SR)	Forward	12	Forward	4
6	HardRight (HR)	Forward	15	Reverse	8

Table 2: Direction and Speed Map as implemented by the PWMCONTROL module

3.10 nnout Module

Like the PWMCONTROL module the NNOUT module takes the Neural Network output and converts it into a pair of PWM and direction outputs. The translation table described in Table 1 is coded using combinational logic and matches the outputs of Table 2.

3.11 seghex Module

The SEGHEX module is used to convert a 4 bit binary input into a 7 bit output suitable for driving a 7 segment LED with inverted logic. The progression of display letters is '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'b', 'C', 'd', 'E', and 'F'. The module is formed using combinational logic and constantly outputs to the LEDs. The SEGHEX module is derived from the SEGOCTAL module of EE 552 Lab 4[3] with 6 new entries added. Figure 3 shows the conversion table, note that the outputs are inverted to match the configuration of the UP1 board.

Input	LED Pattern abcdefg.	Input	LED Pattern abcdefg.
0	00000011	8	00000001
1	10011111	9	00011001
2	00100101	A	00010001
3	00001101	B	11000001
4	10011001	C	01100011
5	01001001	D	10000101
6	01000001	E	01100001
7	00011111	F	01110001

Table 3: 7 Segment LED translation table

3.12 serial Module

Serial output to the base station is done using the SERIAL module. It can write a stream of 8 bit bytes in standard RS-232 serial format (shown in Figure 8)[4]. This is done using the N bit parallel load shift register (See section 3.13) given a 10 bit value "0xxxxxxxx1" and clocked using a CLOCKSCALE module set at the serial baud rate. Since there is only 1 stop bit, 8 data bits and no parity this form of transmission is commonly called "N81". The clock scaler value is selected so that the serial port runs at 9600 baud.

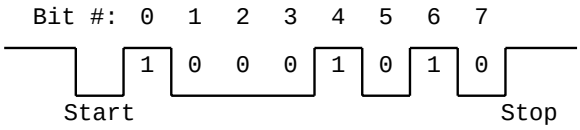


Figure 8: RS-232 Serial Data Format[4]

3.13 shiftout Module

The SHIFTOUT module implements a N bit parallel load shift register with a serial output for use in encoding serial data streams. It has an ENABLE input which can be be used to run the shifter at a speed slower than the chip clock. At the end of the shift sequence the READY line is asserted indicating that new data can be loaded into the shifter. When LATCH is asserted the shifter loads the value into a register and then shifts the register to the left every enabled clock cycle. The bottom bit of the shift register is tied to the output which gives the serial waveform. Figure 9 shows a rough sketches of the main components of the shift register.

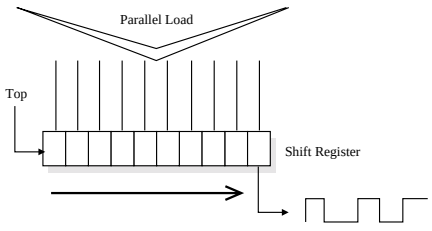


Figure 9: Main Components of the shift out module.

3.14 PWM module

Pulse Width Modulation (PWM) refers to the technique of varying the width of a pulse over time. With respect the DC motors this is frequently used to control the speed of the motor. The percent on time (duty cycle) of the pulse directly relates to the amount of engery imparted to the motor for that cycle. At 100% duty cycle the motor will run at full speed, while at 0% the motor will be fully stopped. The PWM module implements this sort of modulation, the high time is expressed as the number of clock counts the output is high and a fixed max for the counter. The duty cycle can be computed from the expression $\frac{HighTime}{2^{CounterBits}} \times 100\%$ with CounterBits being a generic. Figure 10 shows some example PWM output. For this implemenation the high time is restricted to being 1 less than full on.

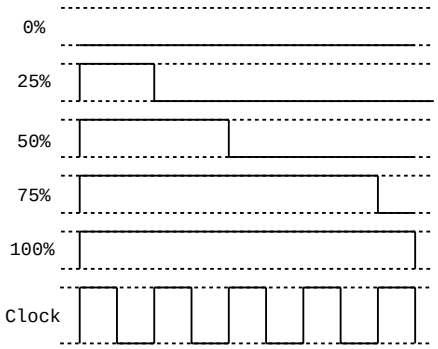


Figure 10: Sample PWM waveforms for a 5 cycle clock

3.15 serialin Module

Data from the base station is retrieved using the SERIALIN module which can decode RS-232 (Figure 8) serial input data. To do this it needs to match the phase of the local 9600 baud clock with the phase of the incoming serial data stream. It does this by pulsing the reset line of the CLOCKSCALE module when the start bit is received. Otherwise it internally implements a simple 8 bit parallel out serial in shit register (the inverse of Figure 9). The state machine implemented by the SERIALIN module is shown in Figure 11.

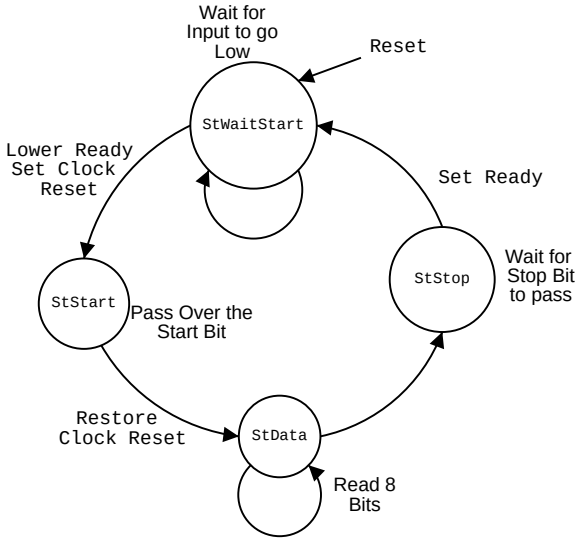


Figure 11: SERIALIN State machine

3.16 clockscale Module

Several of the other modules need rescaled clocks, or periodic (slow) enable pulses to synchronise with external devices. The CLOCKSCALE module provides this. It generates a new clock pulse with a period of SCALE clocks and a 1 clock enable pulse for use by internal logic elements. The rescaled clock is only intended to be used externally. The RESET line can be pulsed to reset the phase of the generated clock. Internal modules are expected to use the system clock and use the ENABLE line to syncronize.

3.17 csadder Module

The CSADDER module implements a single carry save adder with a generic number of bits for the inputs. It was originally developed by Dr. Elliott for an EE552 lab[8]. Figure 12 shows the basic structure of the adder, an array of half adders with their carries unchained, giving the double output. Note that the carry output has been shifted 1 bit to the left so that it is suitable for use in the next stage of the any adder.

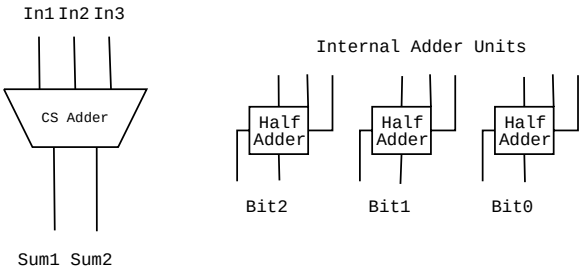


Figure 12: csadder internal structure

3.18 csmult Module

The CSMULT module is a 3 bit by 4 bit carry save multiplier. It performs the normal multiplication step using and gates and then adds those three results using a single 3 input carry save adder. The result is two carry save outputs. One of the two inputs is signed, the other is unsigned. This simplifies the design because the signed inputs are anded and shifted with the unsigned bits then added using a CSADDER module.

3.19 csmultiadder32x7 Module

The CSMULTIADDER32X7 module implements a 32 input 2 output carry save adder. It is a generated module that is constructed by a PYTHON script that generates each layer in the tree. At each tree level the current set of inputs are connected to as many full 3 input adders that can be filled and those inputs are then replaced with the outputs of those adders. This processes is repeated recursively until the entire set of inputs has been accounted for by the adder. Figure 13 shows the tree structure for a 6 input tree, the 32 input tree is simply extended along these lines.

The PYTHON script to generate the adder tree is fairly interesting in itself, it first generates a netlist of connections using the recursive mechanism described above. Each connection is created by adding a 'part' to a list of parts that has a part name, bit width and the node numbers of each of the leads. Two parts are used, an adder module and an sign extender. After this netlist is built up the VHDL is generated from it directly by examining the netlist, instantiating the required bussing and then generating interconnected components based on the netlist.

Another instance of the CSMULTIADDER concept is used in the output layer where it is a 16×3 adder.

3.20 hiddenneuron Module

The actual hidden layer neuron is modeled by this module. It takes as input an array of 16 4 bit numbers and an array of 3 bit signed weight factors for each input. It then multiplies the 16 inputs by the 16 weights and adds them all together and truncates to produce the single bit output that represents the unit step function for the neuron. Mathematically this module implements:

$$Output = u(\sum_i w_i in_i)$$

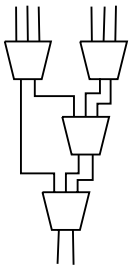


Figure 13: csmultiadder32x7 structural diagram

The weights are signed values and the inputs are unsigned. This module is ment to be embedded into a another module that clocks all 16 weight tables through it to yield the 16 outputs which would then be mapped through another weight table to yeild the 5 bit output of the network. This module was also designed with the aide of PYTHON to work around a bug in MAXPLUS II - see the comment in the source code for details.

3.21 outneuron Module

The OUTNEURON is functionally identical to the HIDDENNEURON except that it takes in 1 bit inputs and 3 bit weights. This means the multiplier can be replaced with a simple *and* gate. The 16 results from the *and* operation are fed into a 16×3 carry save adder tree (see Section 3.19) and then the sign is taken for the result (unit step).

3.22 nnet Module

The NNET module is the neural processor. It combines the HIDDENNEURON, OUTNEURON, and some ROM to evaluate the complete neural network. It uses a 48 (16×3) bit wide LPM_ROM module to store the weights for each seperate neuron. The ROM is connected to both the hidden and output neurons and the selection of weight is based on the address to the ROM. A state machine drives the loading of the two registers so that the entire hidden output is computed before moving on to compute the final output. The contents of the ROM are generated using the MKROM.PY script which takes a table of weights and encodes them as signed 3 bit numbers packed back to back into a single 48 bit value per address. Depending on the number of hidden neurons in use, the weight values for the output neurons may be zero extended by the script.

The state machine has some inbuilt delays for the propogation of the result through the neurons, the length of the delays is determined by the MAXPLUS II timing matrix and is about 200ns. Figure 14(a) shows a structural representation of the module and Figure 14(b) shows the state machine.

3.23 m_rom Module

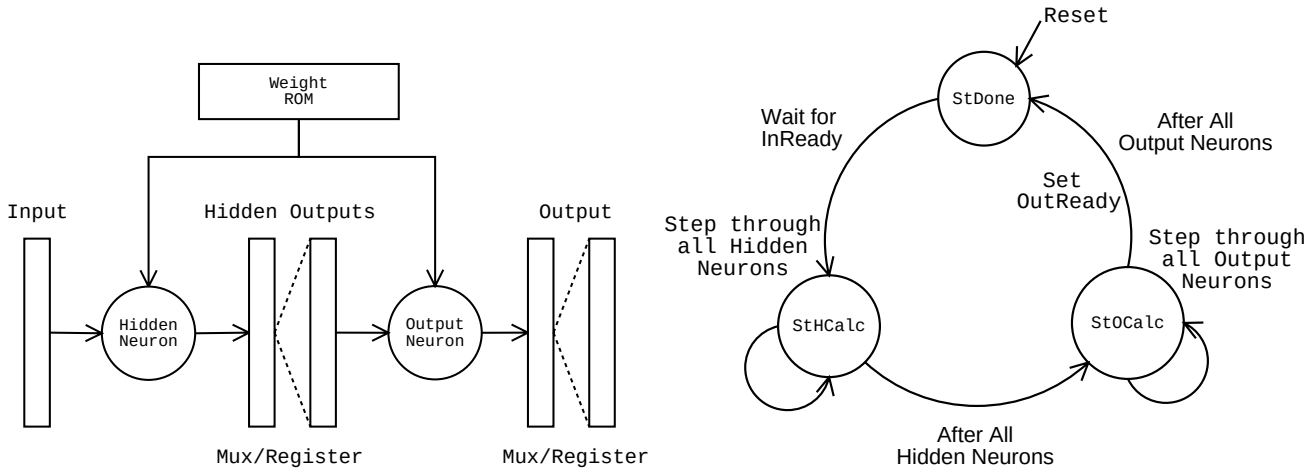
For testing natively in MENTOR GRAPHICS the LPM_ROM module is not available, the M_ROM module provides a substitute. It loads the ROM from a file of bits into a variable array and then mux's that out based on the address. It is used primarily for simulating the functions of the NNET module.

3.24 FPGA Hardware

The ALTERA UP1 FPGA application board was used to protoype the design. Only the two push buttons, two 7 segment LEDs and the FLEX10K chip were used in the design. The other components on the UP1 board were not used. The 'C' connector on the board is connected to a seperate external wire wrap board (see Figure 16) which has the necessary interfacing hardware to connected to the sensor array and motor drivers. Since the FLEX FPGA is SRAM based the programming is not permanent. A separate laptop computer is used to program the FPGA before operation.

3.25 External Hardware

The analog circuitry for the photo-transistor array is shown in Figure 15. A current limiting resistor of 220Ω is placed in series with the infrared led to give a set infrared output. The ground reflected infrared radiation is measured by the photo-transistor side which allows current flow related to the intensity of infrared light received. By placing a 220Ω resistor



(a) Structure

(b) State Diagram

Figure 14: nnet internal structure

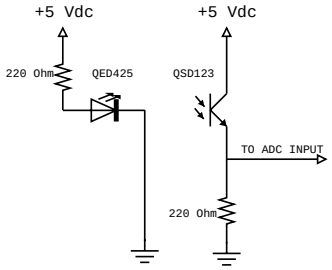


Figure 15: Phototransistor Configuration

on the emitters path to ground, a voltage based on the amount of current flowing through the resistor will be outputted. This voltage can be measured by the ADC which theoretically will range from 0V to 5V assuming perfect light conditions. However, with a black surface, infrared reflection will not be 0% and inversely not 100% for a white surface. This variation will be taken into account when calibrating the circuitry.

Figure 16 shows the schematic layout of the Analog to Digital Converter used with the FPGA. It is setup such that the FPGA has complete control over the chip to get appropriate data from the analog photo-transistor array inputs. Several lines are interconnected for this configuration of the ADC. REF(+) is tied to 5 Vdc, whereas REF(-) is tied to ground. This configures the converter to correlate its 8-bit output values onto the scale from 0 to 5 volts. MUX OUT and COMPARATOR OUT are connected as common to place the ADC0817CNN into the single chip configuration as specified in the application notes from National Semiconductor[1]. This connection prevents the chip from trying to communicate with other cascaded and multiplexed ADCs.

The serial interface was implemented around the standard MAX232 IC. The plans for the circuitry were referenced from Beyond Logic `http://www.beyondlogic.org`[4]. This circuit takes TTL logic levels of 0 or 5V and converts it to RS 232 levels of -12 or 0 V levels. The connection of the FPGA to the PC is achieved through the MAX232 which allows logic lines from the FPGA to communicate with the Rx and Tx lines on the PC serial port.

Figure 16 shows the complete configuration of the external wire wrap board which contains all of the interface circuitry for the design.

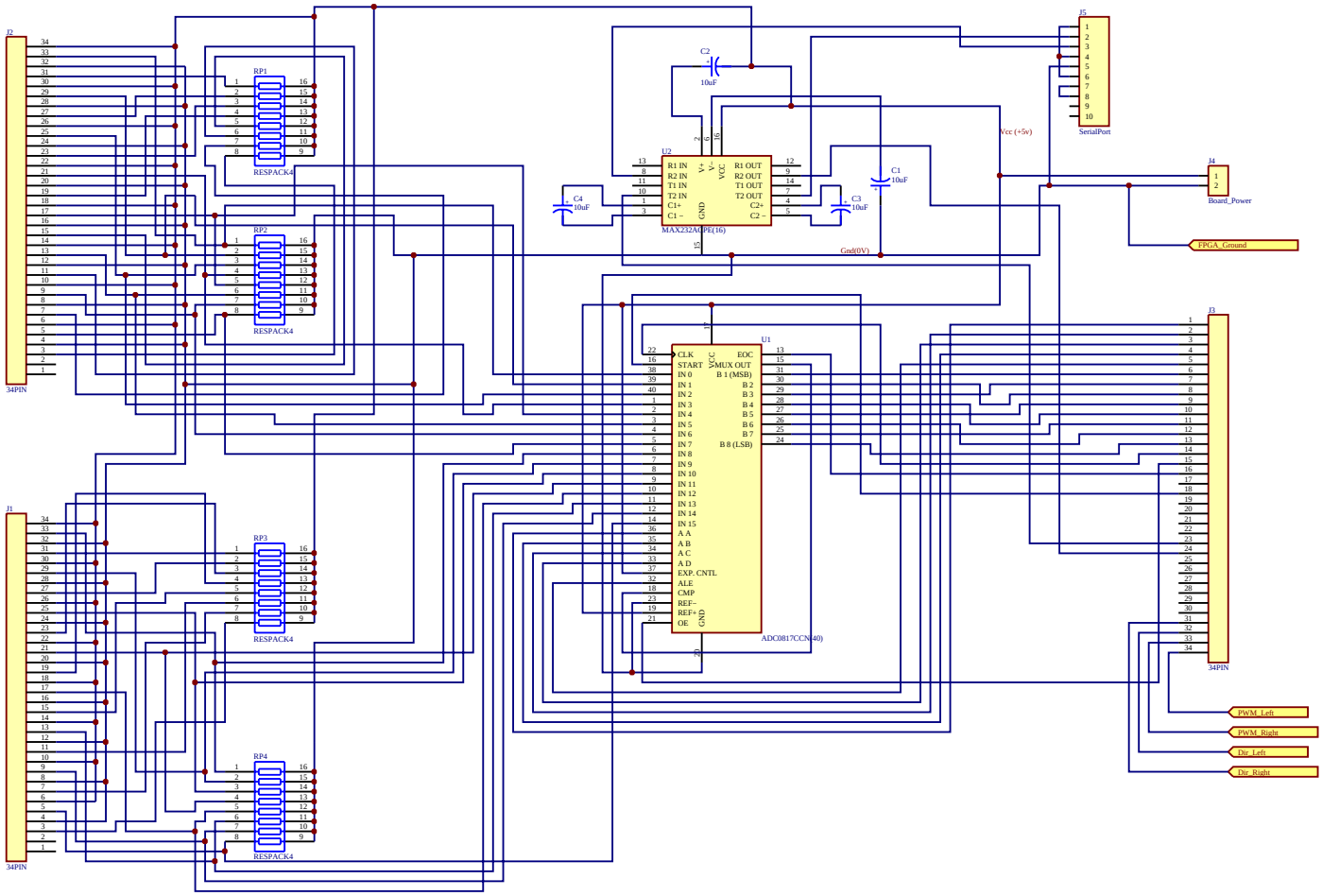


Figure 16: Complete Wire Wrap Circuit Diagram

3.26 Displayer Data Collection Program

A C program called DISPLAYER is used to collect data from the FPGA over the serial port and display it in an X window in real time. The DISPLAYER program uses the ARVP imaging library[14] and GTK wrapper to display the collect bitmap on the screen. It also logs each recieved image to stdout and also records which keystroke is being used. It is structured as 3 independent threads of control. The first is the main thread which reads each image in from the serial port and syncronizes with the FPGA. The second is a serial relay thread which takes keystrokes from stdin and relays them to the FPGA and the final thread is a GTK maintenance thread which runs the main GTK event loop.

The ARVP imaging library provides routines for display and rescaling of the collected images. On the screen the image is scaled up 20 times so that it is easy to read.

4 IO Signals

Summarised in each of the structural diagrams (such as Figure 3) are the external ports available to each entity. The tables here are transcribed from those diagrams, but include the pin assignments on the chip for reference. Table 4 shows the IO connection table for the entire design. The wire wrap board expects the connections in this order and all of the testing modules have been coded to match.

Name	Chip Pin	UP1 Pin	Description
ResetBtn	29	FLEX-PB2	Pushbutton input for Reset
SerIn	190	24(c)	Serial data receive
SerOut	200	25(c)	Serial data transmit
Led1[7:0]	[14-11,9-6]	FLEX_DIGIT1	Upper Nibble LED
Led2[7:0]	[25-23,21-17,14-11]	FLEX_DIGIT2	Lower Nibble LED
ADC_SC	199	16(c)	ADC Start Conversion
ADC_OE	200	18(c)	ADC Output Enable
ADC_ALE	217	17(c)	ADC Address Latch Enable
ADC_Clock	203	19(c)	ADC Clock
ADC_EOC	201	15(c)	ADC End of Conversion
ADC_Mux[3:0]	[221,220,222,219]	47,46,48,45(c)	ADC Channel Select
ADC_Data[7:0]	[218,215,214,208 207,206,204,202]	35,37,38,39, 40,41,42,44(C)	ADC Data Input
PWMLeft	109	15(b)	Left Wheel PWM
PWMRight	110	16(b)	Right Wheel PWM
DirLeft	111	17(b)	Left Wheel Direction Bit
DirRight	113	18(b)	Right Wheel Direction Bit
clock	23	CLK	System Clock

A

Table 4: Training IO Table (see Figure 3)

5 Experimental Results

Testing of the photo transistors using the ADC with no compensation shows that the lower 4 bits of the ADC output are significant, as the intensity of the output can be fairly accurately measured. It is quite possible to determine when a hand is reflecting the infrared light. By using a sheet of white paper it was found that the highest possible output from the A/D is 0x0F. It was also found that the sensor can work fairly well up to about 0.5" off the reflecting surface. Thus the sensor array has been mounted at 'sniffing' distance to the ground (about 0.5").

6 FPGA Requirements

Table 5 summarises the measured requirements of the various modules in the design. The overall design uses 82% of the chip and all 6 EABs for weight ROM. The requirements could be reduced by modifying the implementation to not use adder trees, but instead clock data through a single adder multiple times.

Module	# of logic cells	% of chip used
adc	27	2%
adctest	174	15%
clockscale	6	1%
serial	35	3%
serialin	51	4%
serialtest	186	16%
serialtest2	168	14%
shiftout	22	1%
train	299	25%
button	48	4%
pwmcontrol	14	1%
seghex	19	1%
pwm	10	0%
csmultiadder32x7	488	38%
csmultiadder16x3	114	9%
hiddenneuron	770	66%
outneuron	173	15%
nnet	955	82%

Table 5: Reference table for FPGA requirements.

7 Test Case Index

Each of the test cases is printed later in the report, but this table summaries them all.

train	Mentor Graphics test bench and hardware test
adc	Simulation waveforms
button	Modified simulation waveforms
pwmcontrol	Logic check
seghex	Logic check
serial	Modified simulation waveforms
shiftout	Simulation waveforms
pwm	Simulation waveforms
serialin	Modified simulation waveforms
clockscale	Simulation waveforms
csadder	Simulation waveforms, Complete Test Bench
csmult	Simulation waveforms, Complete Test Bench
adc interface	Hardware test
serial out	Hardware test
serial in	Hardware test
csmultiadders	Random Test Bench
hiddenneuron	Random Test Bench
outputneuron	Random Test Bench
neural net	Random Test Bench

- *Simulation waveforms* indicates the component was simulated in MaxPlus II and the annotated waveform output was included in the report.
- *Modified simulation waveforms* indicates that some aspect of the circuit had to be adjusted for it to be simulated, usually a clock scale factor reduced to a small number
- *Logic check* indicates that the combinational logic inputs and outputs were verified against the expected values.
- *Hardware test* indicates the module was verified independently on the physical hardware.
- *Random Test Bench* indicates that the module was test benched under Mentor Graphics using a random selection of test data and some border cases. This was not done using extracted timing information, it is only a logic check.

8.2 adc Module

Since the ADC generates a slower clock to run the ADC chip, this scale factor was changed to be 2 cycles for testing. Figure 17 shows the operational cycle of the ADC, visiting all possible states (see Figure 7). The simulation was run for several cycles to show the repeating nature of the ADC. Using the registered performance meter the maximum speed of the circuit was found to be 65.35MHz. Testing of the adc module was also done in hardware in Section 11.8.1.

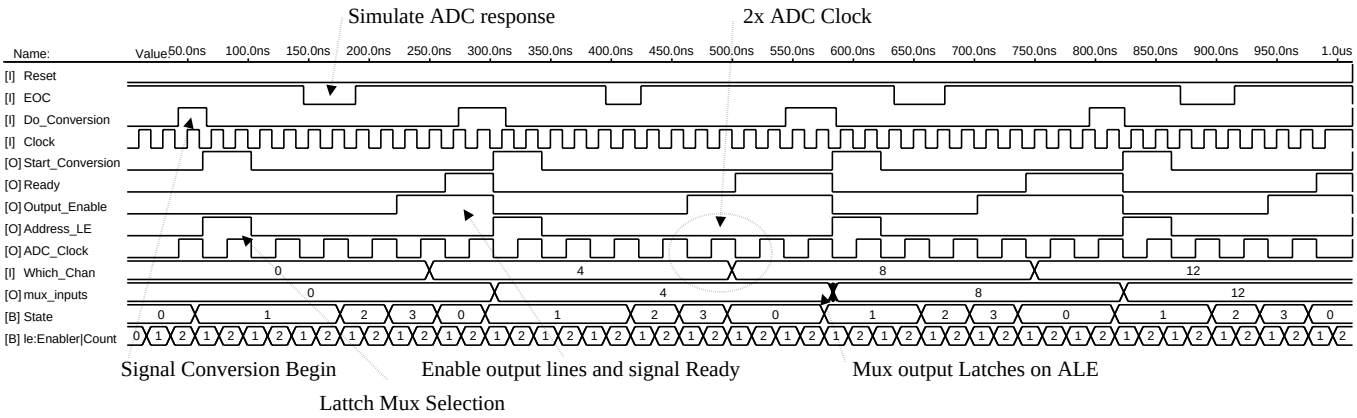


Figure 17: adc module simulation waveforms

8.3 button Module

For testing the button debouncer, the stability constant was reduced to 6 so that the module could be realistically tested in MaxPlus II. The output waveform shows the inverted logic and the requirement for a 6 cycle stability. Using the registered performance meter in MaxPlus II the max speed of the circuit was found to be 25.18MHz. The button debouncer was also extensively tested in the physical hardware tests section (11.8). Figure 18 shows the basic waveforms for the button debouncer.

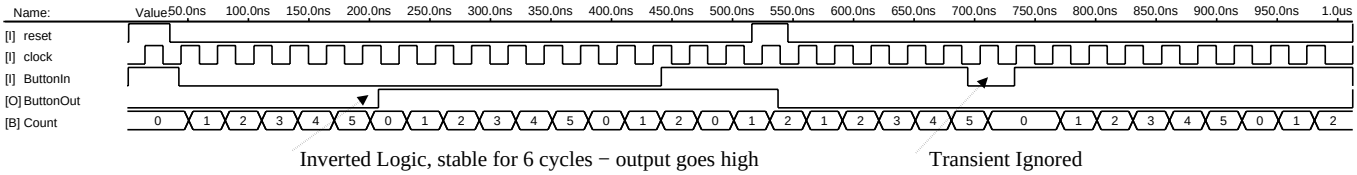


Figure 18: button module simulation waveforms

8.4 pwmcontrol Module

Since the PWMCONTROL module is composed purely of combinational logic the only testing required is to ensure the input matches the required output from Table 2. Each required input was tested to see that it generated the expected output, as well as a extraneous output to show that it generates the same as '5' (stop). The worst case delay was measured on the waveform to be approximately 20ns. Using the MaxPlus II delay matrix the worst case delay was found to be 20.9ns which is 47.8MHz.

8.5 seghex Module

The SEGHEX module is composed purely of combination logic. All that needs to be verified is that the correct outputs are mapped to the correct inputs and confirm the worst case timing. Figure 20 shows the results for all possible inputs and the worst case timing delay. Also, using the MaxPlus II delay matrix it was found that the worst case delay performance was 25.7ns which is 38.9MHz. During testing of the serial output (Section 11.8.2) the SEGHEX module was confirmed to work in

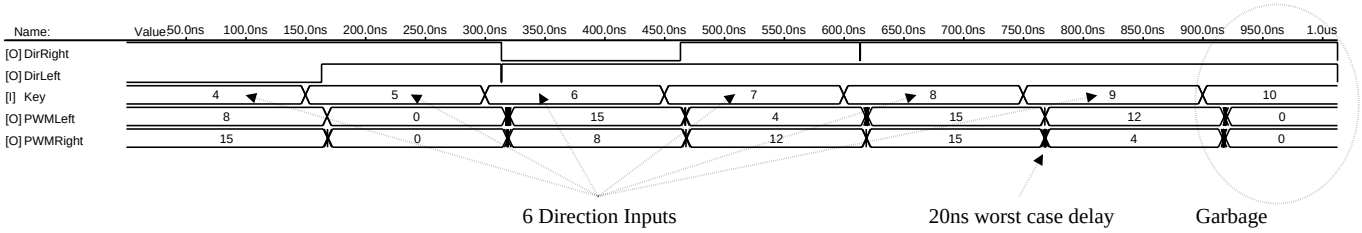


Figure 19: pwmcontrol module simulation waveforms

hardware. The single value step of the serial output tester ran through every single possible display value, and it was visually confirmed that the correct segments were lit.

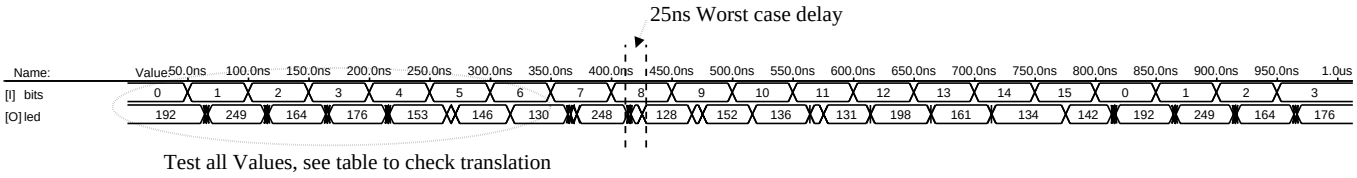


Figure 20: seghex module simulation waveforms

8.6 serial Module

Since the *serial* module is an extremely slow process, the serial baud divisor was set to 2 for testing. This makes the serial output process visible. Figure 21 shows the output for the transmission of '10101010', which matches the required output from Figure 8. The simulation waveforms emulate the external state machine in the SERIALTEST module, they wait for READY to be asserted, then raise LATCH and then wait for READY to go low so LATCH can be lowered. Serial transmission requires no further intervention. In Figure 21 it can also be seen that the period of a '1' pulse is 2 clock cycles, which is what the serial divisor was set to for testing. The registered performance meter measures the design at 45.04 MHz. Testing of the SERIAL module was also done in hardware in Section 11.8.2.

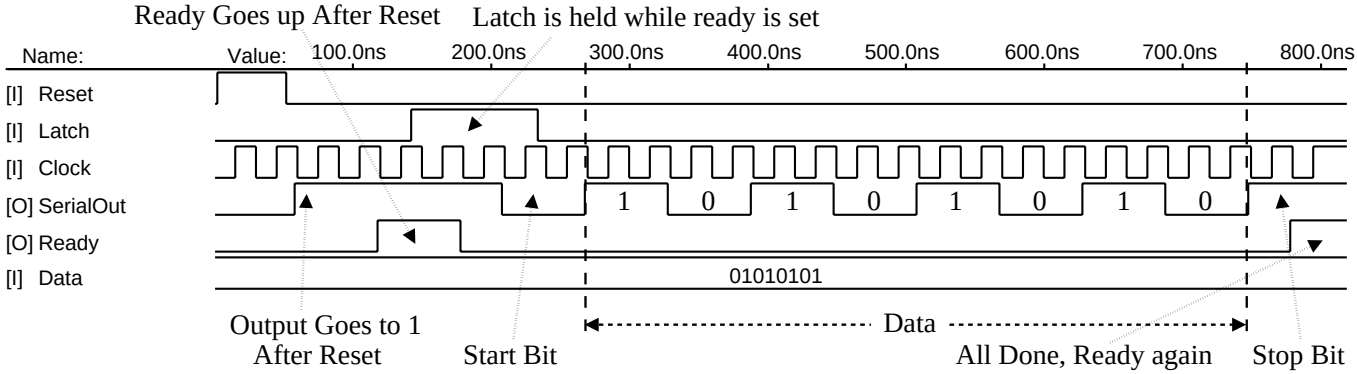


Figure 21: serial module simulation waveforms

8.7 shiftout Module

The serial output shifter was tested by loading two values and shifting them out. The values were selected so that the direction of the shift is apparent. This module also received detailed testing as part of the SERIAL module tests above. The test showed the two cycle dead time inherent in this design. This is acceptable because it will be used as a shifter in a serial

port. Registered performance timing in MaxPlus II gave a maximum speed of 65.35 MHz. Figure 22 shows the simulation waveforms for the test.

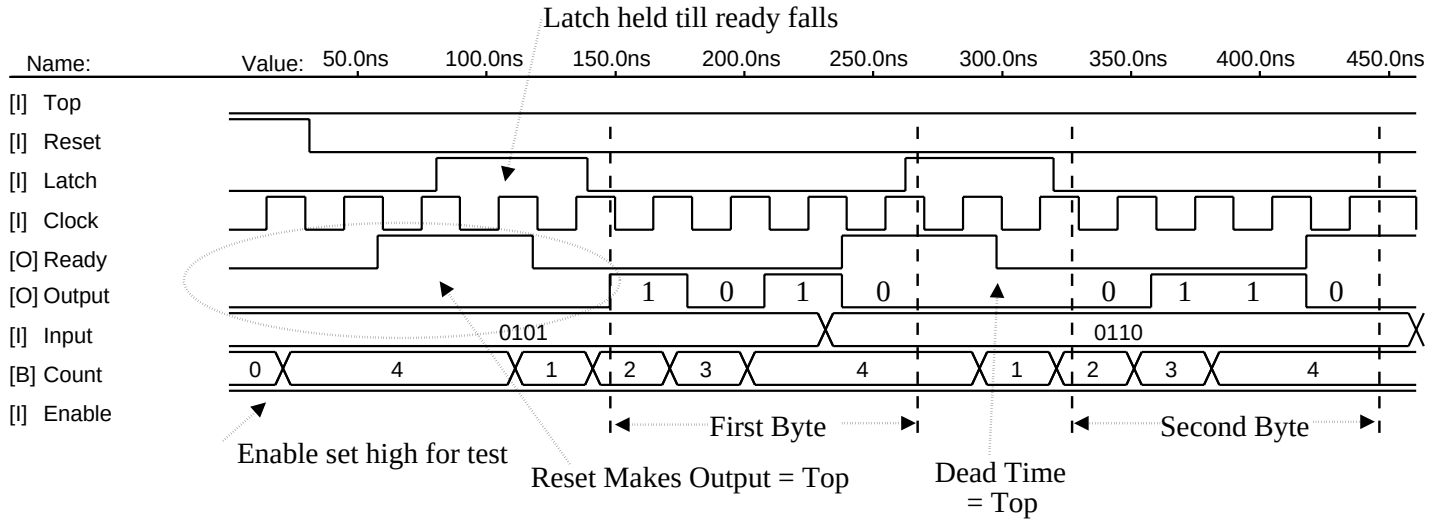


Figure 22: shiftout simulation waveforms

8.8 PWM Module

The two critical aspects of the PWM module, the on time and latching behaviour were tested by simulation with MaxPlus II. In Figure 23 it is shown that the clock cycle is correct for the generic parameter of Bits = 2, while the required behaviour of only switching output frequencies after a cycle is finished is show twice, in the transition from 1 to 2 and 2 to 3. During the hardware testing of the train module the output of the PWM was observed using the oscilloscope to be repeating and free of defects. The registered performance meter for this design gives 96.1 MHz.

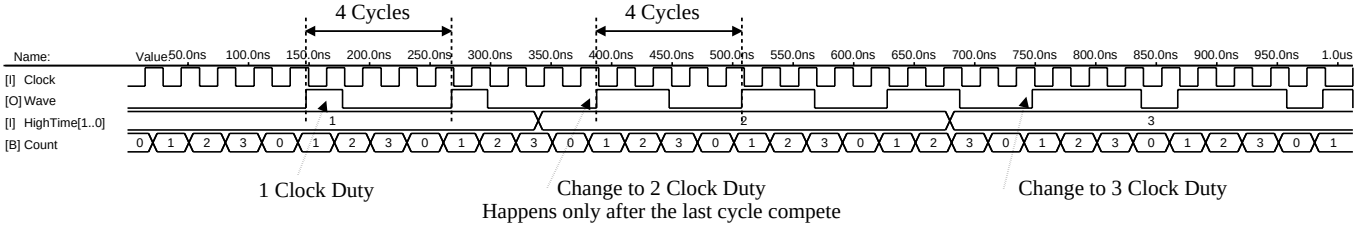


Figure 23: pwm module simulation waveforms (Bits = 2)

8.9 serialin Module

For testing the serialin module the serial baud rate was reset to 2 times clock. This allows the simulation to fit on a single page. The test shown in Figure 24 was to receive a single serial byte and correctly signal reception and load the byte onto the data lines. The byte that was used for testing is '254' or '11111101'. The phase reset of the internal clock scaler can be seen from the internal signal Ck which is the internal reference clock scaler. The registered performance meter shows the circuit's maximum speed is 42.73MHz. Testing of the serialin module was also done in hardware in Section 11.8.3.

8.10 clockscale Module

Two waveform tests were performed using a generic Scale parameter of 4 and 8. These two waveforms were then inspected for correct output. Using the registered performance meter of MaxPlus II it was determined that the maximum clock

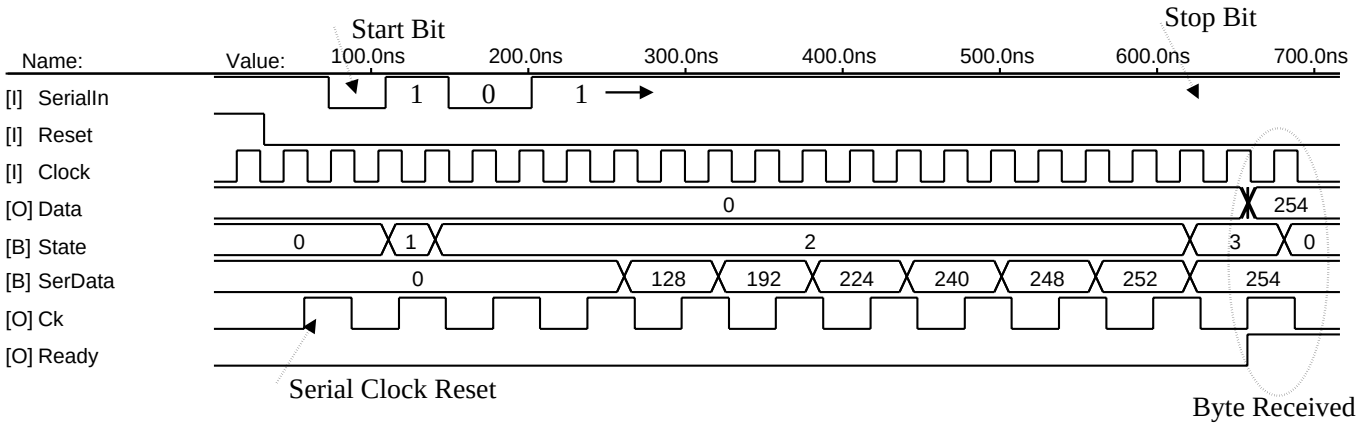
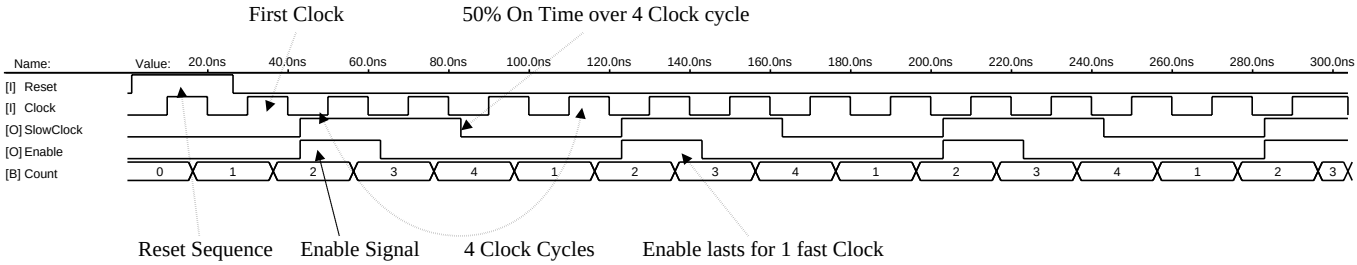
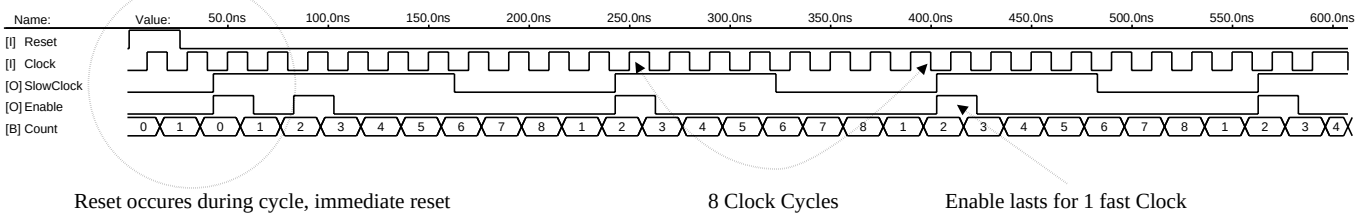


Figure 24: serialin simulation waveforms

frequency the module could handle was 80.64 MHz. Figure 25(a) and 25(b) show the annotated simulation waveforms for the clock scaler. Furthermore the CLOCKSCALE module was verified in a hardware implementation in Sections 11.8.1 and 11.8.2. In particular an oscilloscope was used to measure exactly the output clock waveform (to the ADC) and confirm that it was running at the set period. The top level VHDL contains the constants to set the timer period.



(a) Scale = 4



(b) Scale = 8

Figure 25: clockscale module simulation waveforms

8.11 csadder Module

The carry save adder module was tested by using a selection of values for its three inputs. The results from the two sums were then added by hand and verified against the 3 inputs as correct. Figure 26 shows the generated waveforms. By using the delay matrix it was found the that maximum performance of the circuit is 62.5MHz. The module was also functionally tested in MENTOR GRAPHICS see Section 11.2.

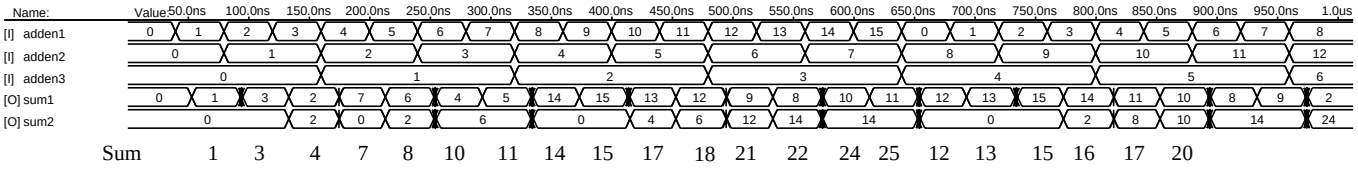


Figure 26: csadder simulations waveforms

8.12 csmult Module

The carry save multiplier module was tested in a manner similar to the csadder module. Several inputs were given and the outputs computed by hand to check for correctness.. Figure 27 shows the generated waveforms. By using the delay matrix it was found that maximum performance of the circuit is 50MHz. This module was also functionally tested in MENTOR GRAPHICS see Section 11.4.

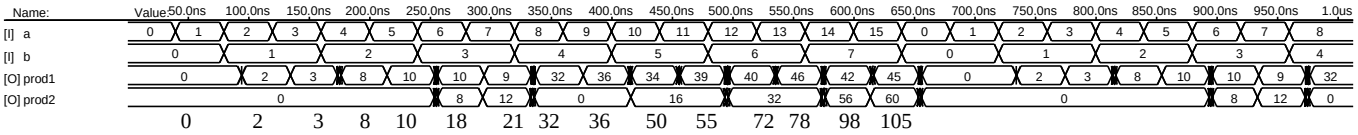


Figure 27: csmult module simulation waveforms

8.13 csmultiadder Modules

The multiple input carry save adder trees were functionally tested under MENTOR GRAPHICS in Section 11.3. For the 32 input adder the latency is approximately 50ns or 20MHz. For the 16 input adder the latency is approximately 37ns or 27MHz. Latency measurements were done using the MAXPLUS II timing matrix.

8.14 hiddenneuron Module

The module was functionally tested in MENTOR GRAPHICS in Section 11.5. The combined Multiplier adder tree in the hidden neuron has a total latency of 105ns or 9.5MHz according to the MAXPLUS II timing matrix.

8.15 outputneuron Module

The module was functionally tested in MENTOR GRAPHICS in Section 11.6. The combined *and* gate adder tree in the output neuron has a total latency of 70ns or 14MHz.

8.16 nnet Module

The module was functionally tested in MENTOR GRAPHICS in Section 11.7. Using the registered performance timing in MAX PLUS II the top speed of the NNET module was found to be 10.59MHz.

9 Design Hierarchy

See Figures 3, 6, 29, 30, and 31 for the hierarchy diagrams. All modules have been effectively compiled and simulated and are belived to be correct. In the design diagrams 'tested' refers to a module that has been simulated and compiled and is belived to be bug free.

10 Source Code Index Index

Each of the files described here is printed in the appendix, 2 pages per sheet.

train.vhd	Top level training module. See Section 3.5 for a description of its operation.
nnrun.vhd	Top level autonomous module. See Section 3.6 for a descirption of its operation.
adc.vhd	Interfacing module for the 0817CCN [1] Analog to Digital converter. See Section 3.7 for a description of its operation.
button.vhd	Button Debouncer for the UP1 board. See Section 3.8 for a description of its operation
pwmcontrol.vhd	Decoder for key strokes to direction control and PWM high times. See Section 3.9 for a description of its operation.
nnout.vhd	Neural network output translation. See Section 3.10 for a description of its operation
seghex.vhd	Displays 4 bit numbers in hexadecimal on the UP1 LEDs. See Section 3.11 for a description of its operation.
serial.vhd	RS-232 serial output. See Section 3.12 for a description of its operation.
shiftout.vhd	N Bit Parallel load serial shifter. See Section 3.13 for a description of its operation.
pwm.vhd	Pulse Width Modulation generator. See Section 3.14 for a description of its operation.
serialin.vhd	RS-232 serial input. See Section 3.15 for a description of it's operation.
clockscale.vhd	Clock scaler. See Section 3.16 for a description of it's operation.
adctest.vhd	Hardware tester for the ADC module. See Section 11.8.1.
serialtest.vhd	Hardware serial output tester. See Section 11.8.2.
serialtest2.vhd	Hardware serial input tester. See Section 11.8.3.
top.vhd	Top level master package, contains component and constant definitions common to the whole project.
makefile	Mentor Graphics compilation makefile.
train_tbench.vhd	train module test bench for Mentor Graphics. See Section 11.1.
csa_tbench.vhd	csadder module test bench for Mentor Graphics. 11.2
csam_tbench.vhd	csmultiadder32x7 test bench for Mentor Graphics. See Section 11.3.
csm_tbench.vhd	csmult test bench for Mentor Graphics. See Section 3.18.
hn_tbench.vhd	hiddenneuron test bench for Mentor Graphics. See Section 11.5.
nnet_tbench.vhd	nnet test bench for Mentor Graphics. See Section 11.7.
on_tbench.vhd	outneuron test bench for Mentor Graphics. See Section 11.6.
dignip-exec.c	C version using floating point of the feed forward network. See Section 3.4 for a description of the c programs.
dignip-train.c	C neural network training program.
dignip-vhdl.c	C version using VHDL-like approximations of the feed forward neural network.
csadder.vhd	Carry save adder . See Section 3.17 for a description of its operation.
csmult.vhd	A single carry save multiplier. See Section 3.18 for a description of its operation.
nnet.vhd	Neural Processor module . See Section 3.22 for a description of its operation.
m_rom.vhd	lpm_rom emulation module. See Secton 3.22 for a description of its operation.
csmultiadder32x7.vhd	The 32 input carry save adder tree. See Section 3.19 for a description of its operation.
csmultiadder16x3.vhd	The 16 input carry save adder tree. See Section 3.19 for a description of its operation.
makebigadder.py	Python script that generates csmultiadder32x7.vhd. See Section 3.19 for a description of its operation.
hiddeneuron.vhd	Top level combinational logic for the hidden layer neuron. See Section 3.20 for a description of its operation.
outneuron.vhd	Top level combinational logic for the output layer neuron. See Section 3.21 for a description of its operation.
makehlneuron.py	Python script that generates hiddenneuron.vhd. See Section 3.20 for a brief description of its operation.
makeolneuron.py	Python script that generates outneuron.vhd.
mkcsamtest.py	Generates complete test vectors for csam_tbench.
mkcsmtest.py	Generates random test vectors for csm_tbench.
mkcsatest.py	Generates complete test vectors for csa_tbench.
mkhntest.py	Generates random test vectors for hn_tbench.
mnnettest.py	Generates random test vectors for nnet_tbench..
mkrandomnn.py	Generates a random set of neural network weights
mkrom.py	Creates .mif files for both Max Plus II and m_rom.
bitmapwindow.cc/h	GTK Bitmap display wrapper [14]

error.cc/h	Error handling routines [14]
serial.cc/h	POSIX Serial port interface [14]
displayer.cc	Remote control base station
image.cc/h	Imaging Routines [14]

11 Test Benches

A semi-standard automatic file oriented test bench framework is used for many of the functional tests. The functional tests all use a file of test vectors and expected results, formatted one test per line. The output is a file of results and expected results and any mismatches are automatically logged using a REPORT statement. Test vectors are typically generated randomly due to the very large test space, this is done using PYTHON scripts to generate some random numbers and compute the expected result. This test bench framework was originally developed in EE552 Lab 5 [13]. Generally, unless otherwise noted, the test benches are not done using extracted timing information. They are strickly functional tests. Timing testing is done all at once at the final top level module.

11.1 train Top Level Module

The TRAIN module was tested using a MENTOR GRAPHICS test bench. The test bench simulated a serial port receiver and the ADC chip. It generated data input from the ADC such that the multiplexor selection was fed back into the data lines. This produced a channel variable output. This output was then converted to a serial RS-232 data stream as it left the train module. The tester integrated a SERIALIN module to decode the serial data stream. Each received serial byte was written to the test log file and 100 bytes were recorded in a test run. The serial bytes were then visually inspected for correctness (the appendix contains a copy of this file). In order to properly test the TRAIN module a few changes were made to the VHDL, the first was to reduce the ADC and Serial clocks to 2 and 4 cycles respectively. The second was to enable a special 'no debounce' mode in the reset button by tying the reset line high - this allows the external reset pulse to propagate through to the sub components.

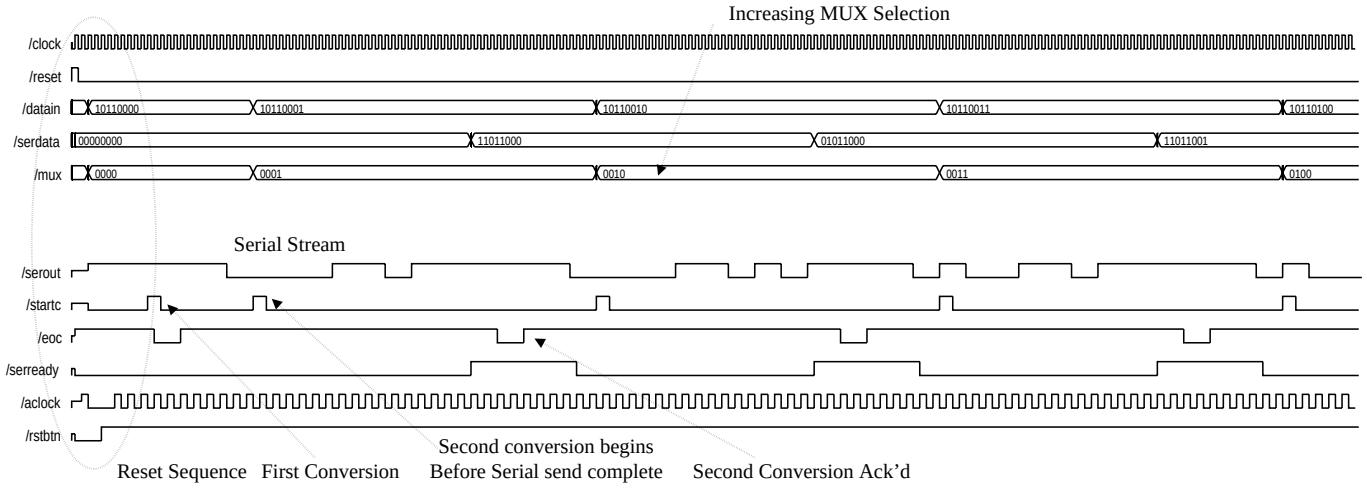


Figure 28: train module simulation waveforms

11.2 csadder Module

The adder was tested comprehensively, all 29791 possible inputs were generated by a PYTHON script and ran through the adder.

11.3 csmultiadder32x7 Module

The multi input adder was testing using 2000 randomly generated inputs, plus two 'min' and 'max' cases. Random testing is used because a 32 input 7 bit adder would require 269599466671506397946670150870196306736371444225405724811036102492 16 separate test cases to test comprehensively.

11.4 csmult Module

The carry save multiplier was comprehensively tested, all possible inputs were computed and verified. This was done using a file oriented test bench and a Python script to generate all possible inputs. Only functional testing was doing using MENTOR GRAPHICS. During testing it was found that the initial carry save implementation was not signed-safe and design changes were made to allow a single signed input.

11.5 hiddenneuron Module

Testing of the completed multiplier/adder combination was done to verify the correct single bit output and the combined operation of the two modules. Random testing using a random set of weights and a random set of inputs was completed. Both the final sum'd output and the truncated single bit output were verified for correctness.

11.6 outputneuron Module

Similarly to the HIDDENNEURON, the OUTPUTNEURON was tested using random weights and random 1 bit inputs. The HIDDENNEURON Python script was used to generate the tests with a different argument.

11.7 nnet Module

The completed network is tested by generating a random set of weights (in the format for M_ROM) and random input values and writing the 5 expected outputs to a log. This is then compared against the expected output file and checked for correctness.

11.8 Physical Hardware Tests

Several tests were conducted on the live hardware, after proper simulation. These tests were intended to confirm that protocols and signalling that the code implemented actually matched what they should be implemented, in particular that our the understanding of the specifications was correct. Each of these tests consisted of a separate top level design driving one of the lower level modules in a specific configuration to allow testing of the module.

11.8.1 ADC Test

The ADC test module allows testing of the ADC under controlled conditions. It uses the two pushbuttons and the 2 LED's on the UP1 board to control the ADC chip. One pushbutton is a global reset, the other takes a single sample from the ADC. The LEDs are hardwired to a the ADC data lines through the SEGHEX decoder module. Figure 29 shows the overall structure of the tester and the external ports. The MUX select is wired to select only channel 0 on the ADC.

Using the ADC test module on the live circuit it was possible to determine that the ADC had a full 5V sensitivity and that it was possible to change the input voltage by exactly 1 bit in the ADC output. An oscilloscope was used to measure the output waveforms and make sure they were within the device tolerances.

11.8.2 Serial Output Test

The SERIALTEST module allows testing of serial transmissions. It uses the two pushbuttons on the UP1 board. One is a system reset, the other is an enter key. Each time the enter key is pressed the value currently shown on the LEDs is transmitted over the serial port and the value increased. A terminal at the other end can receive and display the value on the screen. Values to transmit are limited to 0x30 to 0x3F which is the numeric range of the ASCII table. This makes it easy to display on the terminal. Figure 30 shows the overall structure of the tester and the external ports.

For testing the serial output was connected through a RS-232 level converter[5] so that the terminal could receive the data. An oscilloscope was used the verify that the waveform timings were correct and the signal was clear and undistorted with Figure 8 used as a reference.

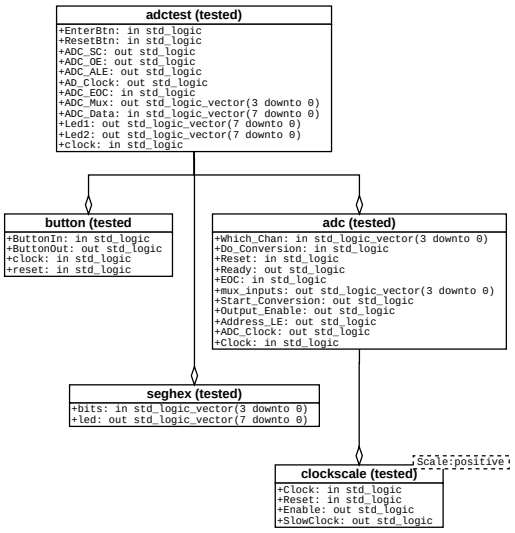


Figure 29: ADC Test Structural Diagram

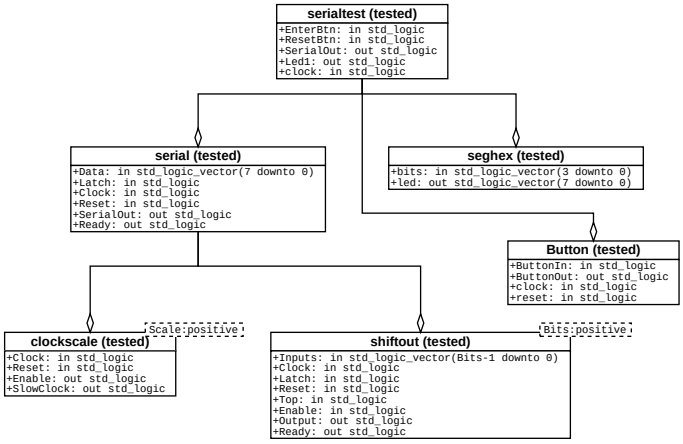


Figure 30: Serial Output Test Structural Diagram

11.8.3 Serial Input Test

The serial input test module was used to show the that serial input works correctly when connected to a real serial port. Each received byte is converted to hex by the SEGHEX module and displayed on the 2 LEDs. One pushbutton is tasked for reset. Otherwise the module constantly operates in this fashion. Typing at the keyboard of the terminal results in the correct ASCII display on the LEDs. Figure 31 shows the overall structure of the tester and the external ports.

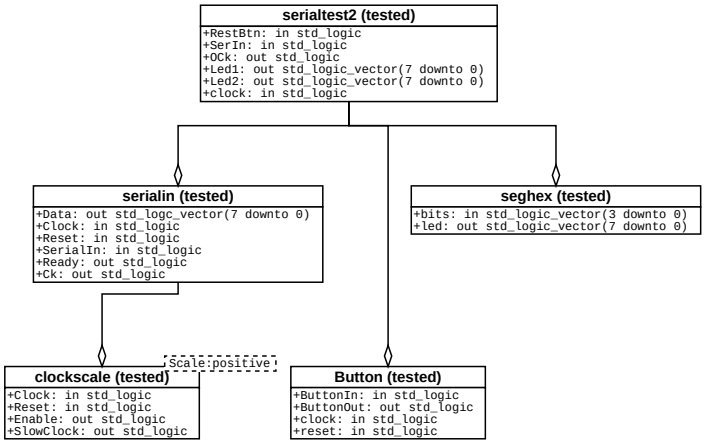


Figure 31: Serial Input Test Structural Diagram

12 Remarks

This report was written by Jason and Darren using a great wealth of UNIX tools. The nice structural diagrams and the state machines were done in a program called Dia `http://www.lysator.liu.se/~alla/dia/` which originally started life as a program to draw UML relationship diagrams, but later grew into a rather nice general purpose program for flowcharts, network diagrams and even some (limited so far, I don't like the results) schematics. The structural diagrams are done using UML object and generic representations while that diamond shaped arrow is the UML symbol for aggregation. This seems to concisely model the structure of the VHDL code in an easy to view format.

The other diagrams were done in a program called Sketch `http://sketch.sourceforge.net/` which is a Corel Draw work alike program for editing vector graphics. To get the MAXPLUS II waveforms into this report we used the pstoedit `http://www.geocities.com/SiliconValley/Network/1958/pstoedit/` program to convert the postscript output from MAXPLUS II into an edit-able vector file. Sketch was then used to annotate this file and create the inset figure!

The report itself was designed in LyX `http://www.lyx.org` which is a nice graphical front end to LaTeX, the scientific typesetting system designed by Donald Knuth.

References

[1] National Semiconductor ADC0817CNN data sheet

[2] Jason Gunthorpe, Darren O'Reilly, EE 552 Lab 6, Winter 2000

[3] Jason Gunthorpe, EE554 Lab 4, Winter 2000

[4] RS-232 interfacing Beyond Logic `http://www.beyondlogic.org/serial/serial1.htm`

[5] MAX232 RS-232 level converter data sheet

[6] VHDL Techniques comp.lang.vhdl FAQ `http://vhdl.org/vi/comp.lang.vhdl/FAQ1.HTML`

[7] Kit Barton and Darren O'Reilly's ARVP Neural Network Tutorial

[8] Dr. Elliott EE552 Lab 4 Carry Save Adder

[9] James Smith and the ARVP will be supplying the mobile robot and the H-Bridge motor drivers.

[10] Darren O'Reilly, David Warkentin and Paul den Boef's EE 563 Neural Image Processor `http://www.ee.ualberta.ca/~oreilly/ee563`

[11] Jeff Woo, lead Bear Cub Designer for the ARVP.

[12] Neual Networks FAQ `ftp://ftp.sas.com/pub/neual/`

[13] Jason Gunthorpe, Darren O'Reilly, EE 552 Lab 5, Winter 2000

[14] Jason Gunthorpe ARVP

Component Datasheets

Figure 32 shows the datasheet for the National Semiconductor ADC0816 8 bit A/D converter. Figure 33 shows the datasheet for the DS14C232 serial level converter, which is functionally equivilant to the MAX232 serial level converter.

June 1999

National Semiconductor

ADC0816/ADC0817
8-Bit µP Compatible A/D Converters with 16-Channel Multiplexer

General Description

The ADC0816, ADC0817 data acquisition component is a monolithic CMOS device with an 8-bit analog-to-digital converter, 16-channel multiplexer and microprocessor compatible control logic. The 8-bit A/D converter uses successive approximation as the conversion technique. The converter features a high impedance chopper stabilized comparator, a 256R voltage divider with analog switch tree and a successive approximation register. The 16-channel multiplexer can directly access any one of 16-single-ended analog signals, and provides the logic for additional channel expansion. Signal conditioning of any analog input signal is eased by direct access to the multiplexer output, and to the input of the 8-bit A/D converter.

The device eliminates the need for external zero and full-scale adjustments. Easy interfacing to microprocessors is provided by the latched and decoded multiplexer address inputs and latched TTL TRI-STATE® outputs.

The design of the ADC0816, ADC0817 has been optimized by incorporating the most desirable aspects of several A/D conversion techniques. The ADC0816, ADC0817 offers high speed, high accuracy, minimal temperature dependence, excellent long-term accuracy and repeatability, and consumes minimal power. These features make this device ideally suited to applications from process and machine control to consumer and automotive applications. For similar performance in an 8-channel, 28-pin, 8-bit A/D converter, see the ADC0808, ADC0809 data sheet. (See AN-258 for more information.)

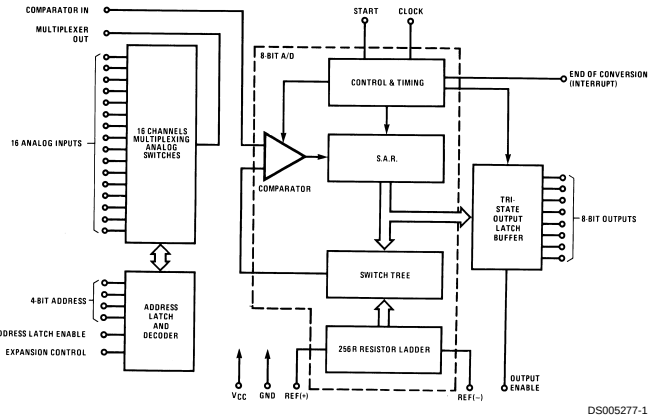
Features

- Easy interface to all microprocessors
- Operates ratiometrically or with 5 V_{DC} or analog span adjusted voltage reference
- 16-channel multiplexer with latched control logic
- Outputs meet TTL voltage level specifications
- 0V to 5V analog input voltage range with single 5V supply
- No zero or full-scale adjust required
- Standard hermetic or molded 40-pin DIP package
- Temperature range −40˚C to +85˚C or −55˚C to +125˚C
- Latched TRI-STATE output
- Direct access to “comparator in” and “multiplexer out” for signal conditioning
- ADC0816 equivalent to MM74C948
- ADC0817 equivalent to MM74C948-1

Key Specifications

■ Resolution	8 Bits
■ Total Unadjusted Error	±½ LSB and ±1 LSB
■ Single Supply	5 V_{DC}
■ Low Power	15 mW
■ Conversion Time	100 µs

Block Diagram



TRI-STATE® is a registered trademark of National Semiconductor Corporation.

 DS005277

Figure 32: National ADC0816 A/D converter Datasheet

National Semiconductor

October 1999

DS14C232
Low Power +5V Powered TIA/EIA-232 Dual Driver/Receiver

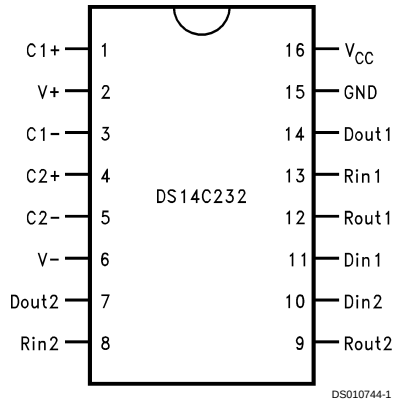
General Description

The DS14C232 is a low power dual driver/receiver featuring an onboard DC to DC converter, eliminating the need for ±12V power supplies. The device only requires a +5V power supply. I_{CC} is specified at 3.0 mA maximum, making the device ideal for battery and power conscious applications. The drivers' slew rate is set internally and the receivers feature internal noise filtering, eliminating the need for external slew rate and filter capacitors. The device is designed to interface data terminal equipment (DTE) with data circuit-terminating equipment (DCE). The driver inputs and receiver outputs are TTL and CMOS compatible. DS14C232C driver outputs and receiver inputs meet TIA/EIA-232-E (RS-232) and CCITT V.28 standards.

Features

- Pin compatible with industry standard MAX232, LT1081, ICL232 and TSC232
- Single +5V power supply
- Low power — I_{CC} 3.0 mA maximum
- DS14C232C meets TIA/EIA-232-E (RS-232) and CCITT V.28 standards
- CMOS technology
- Receiver Noise Filter
- Package efficiency — 2 drivers and 2 receivers
- Available in Plastic DIP, Narrow and Wide SOIC packages
- TIA/EIA-232 compatible extended temperature range option:
  DS14C232T −40˚C to +85˚C
  DS14C232E/J: −55˚C to +125˚C

Connection Diagram



Order Number DS14C232CN, DS14C232CM, or DS14C232TM
See NS Package Number N16E, or M16A

Functional Diagram

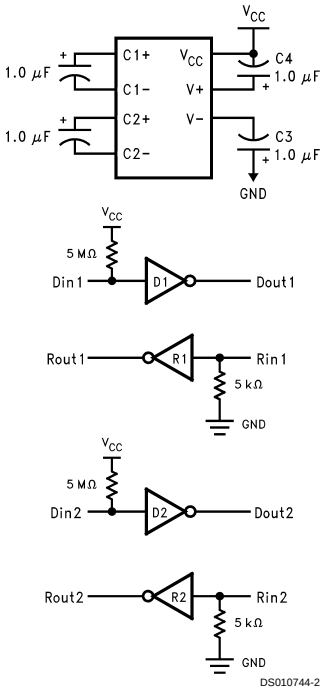


Figure 33: National DS14C232 serial level converter Datasheet