Fall 2011

A rapid-acquisition electrical time-domain reflectometer for analysis of time-variant impedance discontinuities

Joseph Andrew Bishop

Department:

A RAPID-ACQUISITION ELECTRICAL TIME-DOMAIN REFLECTOMETER FOR ANALYSIS OF TIME-VARIANT IMPEDANCE DISCONTINUITIES

by

JOSEPH ANDREW BISHOP

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

2011

Approved by

David J. Pommerenke, Advisor
Genda Chen
James L. Drewniak
Jun Fan

ABSTRACT

A distributed crack sensor has been developed for the measurement of cracks in concrete structures. The sensor is measured using a distributed measurement technique known as electrical time-domain reflectometry (ETDR). ETDR has traditionally been used to measure time-invariant (i.e. unchanging with time) impedance discontinuities, however applications of the sensor in structural failure analysis require measurement of time-variant (i.e. changing with time) impedance discontinuities at rates as high as 10 k measurements per second. ETDR is a suitable measurement technique for these applications since a time-domain reflectometer (TDR) acquisition can be performed in less than 100 μs. Employment of ETDR in these applications, however, requires a TDR that supports measurement rates as high as 10 k measurements per second. Commercial TDRs are not suitable for these applications since their measurement rates are typically less than 10 measurements per second. In order to satisfy the high measurement rates required for these applications, a rapid-acquisition TDR was developed that supports measurement rates as high as 10.1725 k measurements per second. The acquisition rate of the TDR was evaluated by modulating the voltage reflected from a short termination with a voltage variable attenuator. The TDR was able to monitor the reflected voltage at modulation frequencies as high as 1 kHz. The TDR was applied in the monitor of a crack sensor embedded in a bridge column during a shake-table experiment. The TDR was able to monitor the evolution of a crack which formed in the column during the experiment. The operation, design, evaluation, and application of the TDR are discussed herein.

ACKNOWLEDGMENTS

I would like to thank the members of my committee for their time and support. In particular I would like to thank my advisor, Dr. Pommerenke, for his inspiration. He has proven to me time and again that where there is a will there is a way. I would like to thank Dr. Chen for the opportunities he has given me, both in research and in travel. I would also like to thank Dr. Pommerenke and Dr. Chen for their patience throughout the development of the TDR. I would like to thank Dr. Drewniak for emphasizing the importance of the current path at high frequencies. It has helped me more than I ever thought it would. I would like to thank Dr. Fan for agreeing to be a member of my committee on such short notice. I would like to thank Dr. Zoughi for his guidance and inspiration. He has shown me how research can and ought to be conducted.

There are several other individuals who have played a key role in my development of the TDR. I would like to thank Dr. Cox for recommending me to Dr. Pommerenke. I would like to thank Igor Izyumin for introducing me to digital design and for guiding me in the early stages of the TDR development. I would like to thank Yvonne Hardesty for promptly placing the countless orders for components from which the TDR was built.

I would like to thank my friends and family for their encouragement and support throughout my graduate study. I would also like to thank everyone on the sampling scope team at Agilent Technologies for their patience and support during the completion of my thesis.

This work was sponsored by the National Science Foundation (NSF) under grant number CMMI-0409420. The results and opinions expressed in this thesis do not reflect those of the sponsor.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

A distributed, coaxial-cable, crack sensor has been developed for the measurement of cracks in concrete structures [1]-[2]. A photograph of the sensor is shown in Figure 1.1. The sensor is constructed from a ten-gauge, Teflon-insulated, hook-up wire (Dearborn part number 11037) wrapped in a tin-plated, stainless-steel, EMI gasket (Spira part number TNI-.139). The gasket is wrapped tightly around the wire and coated with a thin layer of solder so that there is no separation in the outer conductor of the sensor. When subjected to strain, the thin layer of solder cracks and a separation forms in the outer conductor of the sensor. This separation in the outer conductor creates a detour in the current flowing in the outer conductor which can be modeled as an excess inductance. The excess inductance creates a discontinuity in the characteristic impedance of the sensor which can be measured using ETDR. ETDR has traditionally been used to measure time-invariant (i.e. unchanging with time) impedance discontinuities, however applications of the sensor in structural failure analysis require measurement of time-variant (i.e. changing with time) impedance discontinuities at rates as high as 10 k measurements per second [3]. ETDR is a suitable measurement technique for these applications since a time-domain reflectometer (TDR) acquisition can be performed in less than 100 µs. Employment of ETDR in these applications, however, requires a TDR that supports measurement rates as high as 10 k measurements per second. Commercial TDRs are not suitable for these applications since their measurement rates are typically less than 10 measurements per second. In order to satisfy the high measurement rates required for these applications, a rapid-acquisition TDR was developed that supports measurement rates as high as 10.1725 k measurements per second.

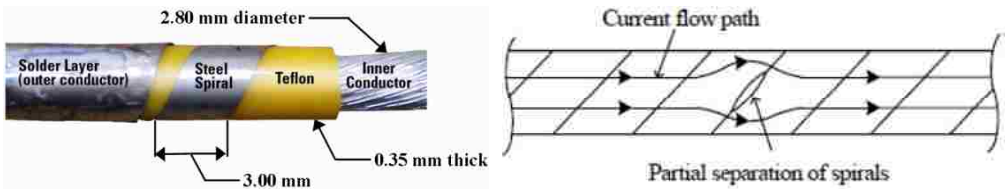

Figure 1.1. Coaxial Cable Crack Sensor

2. SYSTEM

2.1. OPERATION

The operation of the TDR is most easily understood by considering the hypothetical measurement setup of Figure 2.1. The TDR is connected to a transmission line having a length of 10 cm and a characteristic impedance of 50 Ω. The transmission line is terminated in an impedance Z(t) which oscillates at a frequency of 200 Hz such that its corresponding reflection coefficient Γ(t) is $0.1\sin(400\pi t)$. The TDR performs an acquisition every 200 μs so that its acquisition frequency is 25 times the oscillation frequency of the termination impedance Z(t). The reflection coefficient and the values of the reflection coefficient corresponding to the TDR acquisition times are plotted in Figure 2.2 as a function of time. From Figure 2.2 it can be seen that the sinusoidal trend of the reflection coefficient is captured by the values of the reflection coefficient corresponding to the TDR acquisition times. The TDR impulse response acquired at various times is plotted in Figure 2.3. From Figure 2.3 it can be seen that the amplitude of the impulse at 10 cm changes with time and its value is the reflection coefficient of Figure 2.2. The acquired TDR impulse responses are plotted in Figure 2.4 as a function of time. From Figure 2.4 it can be seen that the TDR enables the measurement of impedance discontinuities in both distance and time. The operation of the TDR is analogous to the operation of a video camera. A video camera acquires images at a high rate, while the TDR acquires impulse responses at a high rate.

2.2. STIMULUS

A conventional TDR stimulates the DUT with a voltage step and measures the resulting step response. In some applications only rapid voltage fluctuations associated with high-frequency components of the response are of interest. The fluctuations are often small and require amplification. The extent to which the fluctuations can be amplified is limited by their voltage offset and the vertical range of the TDR. If the vertical range of the TDR is fixed, then the extent to which the fluctuations can be amplified is only limited by their voltage offset. The voltage offset can be cancelled by the addition of another voltage of equal magnitude and opposite polarity, however this

requires knowledge of the both the magnitude and polarity of the voltage offset. A simpler method of removing the voltage offset is to stimulate the DUT with a voltage impulse instead of a voltage step. Since a voltage impulse does not contain the voltage offset contained in a voltage step, the impulse response of the DUT does not contain the voltage offset contained in the step response of the DUT. The TDR differs from a conventional TDR in that it stimulates the DUT with a voltage impulse and measures the resulting impulse response. Impulse and step stimuli are shown in Figure 2.5. From Figure 2.5 it can be seen that the step stimulus contains a large voltage offset which is not contained in the impulse stimulus. The impulse and step responses of a 25 Ω coaxial cable with apertures in its outer conductor are shown in Figure 2.6. The apertures are represented by small rapid voltage fluctuations in both the impulse and step responses. The 25 Ω coaxial cable is represented by a large voltage impulse in the impulse response and a large voltage step in the step response. The impulse and step responses contain the same information, however the step response contains a large voltage offset which is not contained in the impulse response. The amplification limit imposed by the vertical range of the TDR and the voltage offset of the response is illustrated in Figure 2.7 for the impulse and step responses of Figure 2.6. Amplification of the step response results in the destruction of the rapid voltage fluctuations by the lower vertical range limit, whereas amplification of the impulse response results in the preservation of the rapid voltage fluctuations. It should be noted that the processed response type is not necessitated by the measured response type since the impulse and step responses are related by a derivative. The impulse response can be obtained by differentiating and scaling the measured step response. Similarly, the step response can be obtained by integrating and scaling the measured impulse response.

2.3. ACQUISITION

The TDR acquires the impulse response of the DUT using a combination of real-time and equivalent-time sampling. The TDR has two modes of acquisition: single and multiple. In the single acquisition mode the TDR operates as a conventional TDR by performing a single acquisition. In the multiple acquisition mode the TDR acquires

multiple acquisitions at a high acquisition rate. The sampling and acquisition modes of the TDR are discussed in the following subsections.

2.3.1. Sampling. The TDR acquires the impulse response of the DUT using a combination of real-time and equivalent-time sampling. The TDR uses equivalent-time sampling to achieve the sample rate necessary to resolve the high frequency content of the impulse response. Equivalent-time sampling is a method of sampling a repetitive waveform over multiple repetitions by sampling the repetitions at different locations and combining the samples appropriately to reconstruct the waveform. The method of equivalent-time sampling used by the TDR is known as sequential, equivalent-time sampling. In sequential, equivalent-time sampling there is a constant change in the sample location with each repetition of the waveform. Since the sample location is the location of the sample in the waveform, the sample location can be changed by changing the location of the sample or by changing the location of the waveform. The TDR changes the sample location by changing the location of the waveform. The sampled waveform is the impulse response of the DUT. The impulse response of the DUT is the response of the DUT to a voltage impulse generated by the TDR. Since the location of the impulse response is dependent upon the location of the voltage impulse, the location of the impulse response can be changed by changing the location of the voltage impulse. The TDR changes the location of the impulse response by changing the location of the voltage impulse. The location of the voltage impulse is changed by delaying the trigger to the impulse generator with a programmable delay line. The impulse generator is discussed in section 5.2. The programmable delay line is discussed in section 5.1.5. The sequential, equivalent-time sampling used by the TDR is illustrated in Figure 2.8 for the impulse response of Figure 2.6. In addition to equivalent-time sampling, the TDR uses real-time sampling to reduce the acquisition time by taking multiple samples per impulse response. From Figure 2.8 it can be seen that a record length of 3,200 samples would be acquired over 3,200 impulse responses if only one sample were taken per impulse response. If four samples were taken per impulse response, however, a record length of 3,200 samples could be acquired over 800 impulse responses. Thus, the number of impulse responses over which a record length can be acquired is inversely proportional to the number of samples taken per impulse response. In order to reconstruct the impulse

response, the samples taken by the TDR must be combined based on their locations in the impulse response. The locations of the samples in the impulse response are determined by the equivalent times at which the samples were taken. The equivalent times t at which the samples were taken can be calculated from the real-time sample rate R, the equivalent-time sample rate E, the number of samples taken per impulse response S, and the delay values of the programmable delay line d using equation 1. The delay values of the programmable delay line are determined during the horizontal calibration. The horizontal calibration is discussed in section 2.4.

$$\begin{cases} t[n] = \dfrac{1}{R}\left(n - S\left\lfloor \dfrac{n}{S} \right\rfloor - 1\right) - d\left[\dfrac{10^{11}}{E}\left\lfloor \dfrac{n-1}{S} \right\rfloor\right] - d\left[1024 - \dfrac{10^{11}}{E}\right] \\ n = 1, 2, 3, \ldots, \dfrac{1024}{10^{11}} SE \end{cases} \tag{1}$$

2.3.2. Single Acquisition Mode. In the single acquisition mode the TDR operates as a conventional TDR by performing a single acquisition. The TDR performs a single acquisition multiple times if averaging is enabled, however the multiple acquisitions are averaged into a single acquisition before being displayed. In the single acquisition mode the samples acquired from the impulse response of the DUT are written to a buffer in the acquisition controller FPGA. The memory depth in the single acquisition mode is limited by the buffer to 8 kS. After the samples have been written to the buffer they are transferred to the PC over USB. After being transferred to the PC the samples are processed in LabVIEW. After processing the samples LabVIEW plots them in the Control tab of the user interface. The single acquisition mode is the primary mode of acquisition since the multiple acquisition mode is an extension of it.

2.3.3. Multiple Acquisition Mode. In the multiple acquisition mode the TDR acquires multiple acquisitions at a high acquisition rate. In applications where the achievable acquisition rate is much higher the application requires, averaging can be used to attenuate noise in the response. In order to minimize distortion of the response, the time separation of the averaged acquisitions should be minimized by maximizing the acquisition rate. Maximizing the acquisition rate, however, minimizes the time over which the DUT can be monitored since the memory depth of the TDR is fixed. In order

to minimize the time separation of the averaged acquisitions without minimizing the time over which the DUT can be monitored, the multiple acquisition mode is burst oriented as illustrated in Figure 2.9. The acquisitions are acquired in a series of bursts. The acquisitions acquired in each burst are acquired at a constant rate determined by the pulse period, record length, and equivalent-time sample rate. The burst length is the number of acquisitions in each burst and is controlled by the Acquisitions On control on the Control tab of the user interface. The minimum burst length is one acquisition. The burst separation is the number of acquisitions skipped between bursts and is controlled by the Acquisitions Off control on the Control tab of the user interface. The minimum burst separation is one acquisition. The burst duty cycle D_B can be calculated from the burst length L_B and burst separation S_B using equation 2. The burst rate R_B can be calculated from the burst length L_B, burst separation S_B, and acquisition rate R_A using equation 3. It can be seen from equation 3 that the maximum burst rate is half the acquisition rate since the minimum burst length and burst separation are one acquisition. In the multiple acquisition mode averaging is performed by averaging the acquisitions within each burst. The averaged acquisition rate is the acquisition rate after averaging and is indicated by the Averaged Acquisition Frequency (Hz) control on the Control tab of the user interface. The averaged acquisition rate is equal to the burst rate since the acquisitions within each burst are averaged. In the multiple acquisition mode the samples acquired from the impulse response of the DUT are written to the acquisition controller SDRAM. The memory depth in the multiple acquisition mode is limited by the SDRAM to 128 MS. After the samples have been written to the SDRAM they are transferred to the CompactFlash card. After being transferred to the Compact Flash card the samples can be transferred to the PC using a CompactFlash card reader and a hex editor. After being transferred to the PC the samples can be processed in LabVIEW through the Process tab of the user interface. After processing the samples LabVIEW invokes MATLAB to plot them. The multiple acquisition mode is the secondary mode of acquisition since it is an extension of the single acquisition mode.

$$D_B = \frac{L_B}{L_B + S_B} \tag{2}$$

$$R_B = \frac{R_A}{L_B + S_B} \tag{3}$$

2.4. HORIZONTAL CALIBRATION

The time base of the TDR is determined by the delay values of a programmable delay line. Although the delay line is temperature and voltage controlled, a horizontal calibration was implemented to maximize accuracy and enable in-system measurement of the delay values. The horizontal calibration measures the delay corresponding to each delay setting and generates a text file containing the delay settings and values. A portion of the text file generated by the horizontal calibration is shown in Figure 2.10. The delay line was designed to support two modes of operation: normal and calibration. In the normal mode of operation the output of the delay line is the input of the delay line delayed by the delay value. In the calibration mode of operation the output of the delay line is inverted and fed back to its input to form a ring oscillator. The delay values of the delay line are calculated from the oscillation frequencies of the ring oscillator which are measured by a frequency counter in the FPGA. The frequency counter is discussed in section 6.3.8. The oscillation period of the ring oscillator is the reciprocal of its oscillation frequency. The ring delay of the ring oscillator is half its oscillation period since there are two logic transitions per oscillation period. A change in the delay value of the delay line is equal to the resulting change in the ring delay of the ring oscillator since changes in its ring delay are only caused by changes in the delay value of the delay line. The delay values d of the delay line and the oscillation frequencies f of the ring oscillator are related by equation 4, where m is the delay setting of the delay line. The precision of the horizontal calibration can be analyzed by comparing the delay values between multiple calibrations. Ten horizontal calibrations were performed consecutively and the delay values were recorded. The delay values are plotted in Figure 2.11. Since the difference between adjacent delay values is roughly constant and much smaller than the range of delay values, it is convenient to compare the differential delay values between calibrations. The differential delay values δ are related to the delay values d by equation 5, where m is the delay setting of the delay line. The differential delay values are plotted in Figure 2.12. The mean deviations $\overline{A}$ of the differential delay values δ were

calculated using equation 6, where m is the delay setting of the delay line, and are plotted in Figure 2.13. From Figure 2.13 it can be seen that the mean deviations were less than 80 fs. The maximum deviations A_{max} of the differential delay values δ were calculated using equation 7, where m is the delay setting of the delay line and n is the calibration number, and are plotted in Figure 2.14. From Figure 2.14 it can be seen that the maximum deviations were less than 400 fs. The mean relative deviations $\overline{R}$ of the differential delay values δ were calculated using equation 8, where m is the delay setting of the delay line, and are plotted in Figure 2.15. From Figure 2.15 it can be seen that the mean relative deviations were less than 1.6 %. The maximum relative deviations R_{max} of the differential delay values δ were calculated using equation 9, where m is the delay setting of the delay line and n is the calibration number, and are plotted in Figure 2.16. From Figure 2.16 it can be seen that the maximum relative deviations were less than 8 %. It can be seen from Figure 2.13 through Figure 2.16 that the precision of the horizontal calibration is high. The precision of the horizontal calibration can be increased by averaging the delay values between multiple calibrations, however the precision is high enough that it is not necessary.

$$\begin{cases} d[0] = 0 \\ d[m] = \dfrac{1}{2}\left(\dfrac{1}{f[m]} - \dfrac{1}{f[0]}\right) \\ m = 1, 2, 3, \ldots, 1023 \end{cases} \tag{4}$$

$$\begin{cases} \delta[m] = d[m] - d[m-1] \\ m = 1, 2, 3, \ldots, 1023 \end{cases} \tag{5}$$

$$\begin{cases} \overline{A}[m] = \dfrac{1}{N}\displaystyle\sum_{n=1}^{N} \left|\delta[m,n] - \overline{\delta}[m]\right| \\ m = 1, 2, 3, \ldots, 1023 \end{cases} \tag{6}$$

$$\begin{cases} A_{\max}[m] = \max \left|\delta[m,n] - \overline{\delta}[m]\right| \\ m = 1, 2, 3, \ldots, 1023 \\ n = 1, 2, 3, \ldots, N \end{cases} \tag{7}$$

$$\begin{cases} \overline{R}[m] = \dfrac{1}{N}\displaystyle\sum_{n=1}^{N} \dfrac{\left|\delta[m,n] - \overline{\delta}[m]\right|}{\overline{\delta}[m]} \\ m = 1, 2, 3, \ldots, 1023 \end{cases} \tag{8}$$

$$\begin{cases} R_{\max}[m] = \max \dfrac{\left|\delta[m,n] - \bar{\delta}[m]\right|}{\bar{\delta}[m]} \\ m = 1,2,3,\ldots,1023 \\ n = 1,2,3,\ldots,N \end{cases} \tag{9}$$

2.5. VERTICAL CALIBRATION

A vertical calibration can be performed to remove the effects of the TDR and interconnects on the impulse response of the DUT. The vertical calibration consists of two operations: normalization and de-embedding. Normalization and de-embedding are discussed in the following subsections.

2.5.1. Normalization. The impulse response of a 50 Ω termination measured by an ideal TDR contains no reflections, however the impulse response of a 50 Ω termination measured by a real TDR contains small reflections caused by impedance mismatches. The impedance mismatches include the impedance mismatch of the measurement port and impedance mismatches within the TDR. Additional impedance mismatches are created if interconnects are used to connect the 50 Ω termination to the TDR. The small reflections caused by the impedance mismatches are present in the impulse response of every DUT since they are present in the impulse response of a 50 Ω termination. Since the small reflections are present in the impulse response of every DUT, they can be removed from the impulse response of a given DUT by subtracting the impulse response of a 50 Ω termination from it. This process is referred to as normalization since the impulse response of a DUT is normalized to the impulse response of a 50 Ω termination. The normalized and unnormalized impulse responses of a 50 Ω termination are plotted in Figure 2.17. From Figure 2.17 it can be seen that normalization removed the small reflections in the unnormalized impulse response of the 50 Ω termination. The normalized and unnormalized impulse responses of a short termination are plotted in Figure 2.18. From Figure 2.17 and Figure 2.18 it can be seen that the small reflections in the unnormalized impulse response of the 50 Ω termination are present in the unnormalized impulse response of the short termination. From Figure 2.18 it can be seen that normalization removed the small reflections in the unnormalized impulse response of the short termination which were present in the unnormalized

impulse response of the 50 Ω termination. The normalized and unnormalized impulse responses of an open termination are plotted in Figure 2.19. From Figure 2.17 and Figure 2.19 it can be seen that the small reflections in the unnormalized impulse response of the 50 Ω termination are present in the unnormalized impulse response of the open termination. From Figure 2.19 it can be seen that normalization removed the small reflections in the unnormalized impulse response of the open termination which were present in the unnormalized impulse response of the 50 Ω termination. From Figure 2.17 through Figure 2.19 it can be seen that the unnormalized impulse responses of the open and short terminations also contain reflections which are not present in the unnormalized impulse response of the 50 Ω termination. Normalization does not remove these reflections since they are not present in the unnormalized impulse response of the 50 Ω termination. These reflections are part of the impulse response of the TDR and can be removed from the impulse response of the DUT by de-embedding the impulse response of the DUT from the impulse response of the TDR. De-embedding is discussed in section 2.5.2.

2.5.2. De-embedding. The impulse response of a DUT measured by an ideal TDR is the impulse response of the DUT, however the impulse response of a DUT measured by a real TDR is the impulse response of the DUT convolved with the impulse response of the TDR. In order to obtain the impulse response of the DUT, the measured impulse response of the DUT must be deconvolved with the impulse response of the TDR. This process is referred to as de-embedding since the impulse response of the DUT is de-embedded from the impulse response of the TDR. Before the measured impulse response of the DUT can be deconvolved with the impulse response of the TDR, the impulse response of the TDR must be obtained. The impulse response of the TDR can be obtained by measuring an open termination, however a true open termination cannot be achieved in practice because of fringing and radiated fields. The impulse response of the TDR can also be obtained by negating the measured impulse response of a short termination since the reflection coefficient of a short termination is the negative of the reflection coefficient of an open termination. Once the impulse response of the TDR has been obtained it can be deconvolved with the measured impulse response of the DUT. The de-embedding process can be simplified by transforming the time-domain

deconvolution into a frequency-domain division using the fast Fourier transform (FFT). The result of the frequency-domain division is the frequency response of the DUT. The impulse response of the DUT can be obtained from the frequency response of the DUT using the inverse fast Fourier transform (IFFT). The impulse response of the DUT is the response of the DUT to an ideal impulse. An ideal impulse has unlimited bandwidth, however a real TDR has limited bandwidth. The limited bandwidth of a real TDR will result in an extremely noisy impulse response, unless the bandwidth of the frequency response of the DUT is limited with a filter. The impulse response resulting from the filtered frequency response of the DUT is the impulse response of the DUT convolved with the impulse response of the filter. This implies that the de-embedded stimulus is the impulse response of the filter since the response of a linear system to a stimulus is the impulse response of the system convolved with the stimulus. The TDR filters the frequency response of the DUT with a Gaussian filter so that its de-embedded stimulus is a Gaussian impulse. The bandwidth of the filter is determined by the rise time of its step response which is controlled by the Rise Time (ps) control on the Control tab of the user interface. The extreme noise resulting from the limited bandwidth of a real TDR can be avoided with filtering, however there are other problems which can result from the frequency-domain division if the value of the denominator is too small. If the values of the numerator and denominator are zero, then the result of the division is indeterminate. If the value of the numerator is nonzero and the value of the denominator is too small, then the result of the division is too large to represent. The division problems can be avoided by truncating the frequency response of the TDR at the first value which is too small. If the frequency response of the TDR is truncated, then the embedded frequency response of the DUT must also be truncated. The truncated embedded frequency response of the DUT can be divided by the truncated frequency response of the TDR to obtain the truncated frequency response of the DUT. Truncating the frequency response of the DUT decimates the impulse response of the DUT, however the decimated impulse response of the DUT can be reinterpolated. A simplified MATLAB implementation of the de-embedding algorithm used by the TDR is shown in Figure 2.20. The embedded and de-embedded impulse responses of an open termination are shown in Figure 2.21 and Figure 2.22. The embedded and de-embedded impulse responses of a short

termination are shown in Figure 2.23 and Figure 2.24. The embedded and de-embedded step responses of an open termination are shown in Figure 2.25 and Figure 2.26. The embedded and de-embedded step responses of a short termination are shown in Figure 2.27 and Figure 2.28. Further information on de-embedding can be found in [8]-[9].

2.6. PROCESSING

2.6.1. Peeling. The reflection coefficient of a discontinuity in the step response can be calculated from the amplitudes of the incident and reflected waves. The reflection coefficient of the first discontinuity is easily calculated since the amplitude of the incident wave is the amplitude of the pulse stimulus and the amplitude of the reflected wave is the amplitude of the measured reflected pulse. The reflection coefficients of subsequent discontinuities, however, become increasingly difficult to calculate since the amplitudes of their incident and reflected waves are dependent on the preceeding discontinuities. The TDR overcomes this difficulty by employing a discrete, inverse-scattering algorithm known as the peeling algorithm. The peeling algorithm evaluates the N point step response as the response of an N-segment, lossless transmission line to an ideal step. The peeling algorithm is discussed in detail in [4]. A simplified MATLAB implementation of the peeling algorithm used by the TDR is shown in Figure 2.29. The peeling algorithm was evaluated on the stepped-impedance, microstrip transmission line shown in Figure 2.30. The impedance profile with and without peeling is shown in Figure 2.31. From Figure 2.31 it can be seen that although the impedance of the first discontinuity is the same with and without peeling, the impedances of the subsequent discontinuities are significantly different with and without peeling. It should be noted that peeling does not account for dielectric and skin-effect losses which is why the second 95 Ω to 25 Ω transition in the impedance profile is slower than the first.

2.6.2. Windowing. The impulse and step responses of a DUT contain reflections caused by impedance discontinuities along the length of the DUT. The locations of the reflections correspond to the locations of the impedance discontinuities. If the impedance discontinuities cause multiple reflections, however, ghost reflections may appear at locations beyond the physical length of the DUT. These ghost reflections are shown in Figure 2.32 for the 25 Ω coaxial cable with apertures of section 2.2. The

TDR incorporates a windowing operation that can be used to remove ghost reflections by specifying the location of the end of the DUT. The location of the end of the DUT is specified by the location of the window marker in the impulse and step responses of the DUT. The windowing operation removes ghost reflections from the impulse and step responses by multiplying the portion of the impulse response to the right of the window marker location by zero. The impulse response of the 25 Ω coaxial cable with apertures is shown in Figure 2.33 with and without windowing. The step response of the 25 Ω coaxial cable with apertures is shown in Figure 2.34 with and without windowing. The windowed impulse and step responses of the DUT are the impulse and step responses which would be obtained if the DUT were terminated at the marker location with a matched impedance. If ghost reflections are not removed, they can interfere with the de-embedding operation resulting in a degradation of the processed response. Degradation of the step response resulting from ghost reflections interfering with the de-embedding operation is shown in Figure 2.35. The interference of ghost reflections with the de-embedding operation is a result of their extending the duration of the embedded impulse response. Extending the duration of the embedded impulse response results in aliasing of the de-embedded impulse response if the number of samples in the de-embedded frequency response is insufficient. Aliasing of the de-embedded impulse response can be avoided by zero padding the embedded impulse response or by removing the ghost reflections from the embedded impulse response with windowing. Windowing is effectively zero padding since the portion of the embedded impulse response to the right of the window marker location is multiplied by zero.

2.6.3. Smoothing. The TDR offers two methods of controlling the bandwidths of the impulse and step responses: de-embedding and smoothing. The de-embedding operation controls the bandwidths of the impulse and step responses by controlling the bandwidth of the Gaussian filter applied to the de-embedded frequency response. The de-embedding operation is discussed in section 2.5.2. The smoothing operation controls the bandwidths of the impulse and step responses by controlling the number of samples in a moving average applied to the impulse and step responses. The embedded impulse response of a 41 Ω termination is plotted in Figure 2.36. From Figure 2.36 it can be seen that the embedded impulse response contains high frequency noise in addition to the

impulse reflected from the 41 Ω termination. The smoothing operation was applied to the embedded impulse response of Figure 2.36 and the smoothed impulse response is plotted in Figure 2.37. From Figure 2.36 and Figure 2.37 it can be seen that the smoothing operation acted as a low-pass filter by attenuating the high frequency noise. The bandwidth of the smoothing filter is a function of the equivalent-time sample rate and the number of samples in the moving average. The bandwidth of the smoothing filter can be numerically determined by smoothing an ideal digital impulse and calculating its frequency response. The bandwidth of the smoothing filter as a function of the number of samples in the moving average is plotted in Figure 2.38 for equivalent-time samples rates of 12.5 GS/s, 25 GS/s, 50 GS/s, and 100 GS/s. The bandwidth of the smoothing filter can be expressed as a percentage of the equivalent-time sample rate by dividing by the equivalent-time sample rate and multiplying by 100. The bandwidth of the smoothing filter as a percentage of the equivalent-time sample rate is plotted in Figure 2.39. From Figure 2.38 and Figure 2.39 it can be seen that the bandwidth of the smoothing filter is directly proportional to the equivalent-time sample rate and inversely proportional to the number of samples in the moving average. From Figure 2.38 and Figure 2.39 it can also be seen that the constant of proportionality is approximately 0.465. The bandwidth B of the smoothing filter can be calculated from the equivalent-time sample rate E and number of samples N in the moving average using equation 10.

$$B = 0.465\frac{E}{N} \tag{10}$$

2.6.4. Excess Reactance. The TDR incorporates an excess reactance calculator which can be used to approximate the excess reactance of inductive and capacitive impedance discontinuities in the step response of a DUT. If the voltage reflected from an impedance discontinuity is positive as shown in Figure 2.40, then the impedance discontinuity is inductive and is modeled as a series inductance as shown in Figure 2.41. The response of the series inductance model is shown in Figure 2.43. The series inductance L can be approximated from the response of the series inductance model using equation 11. If the voltage reflected from an impedance discontinuity is negative

as shown in Figure 2.40, then the impedance discontinuity is capacitive and is modeled as a shunt capacitance as shown in Figure 2.42. The response of the shunt capacitance model is shown in Figure 2.44. The shunt capacitance *C* can be approximated from the response of the shunt capacitance model using equation 12. It should be noted that the reflection coefficient ρ in equations 11 and 12 is determined by the incident and reflected waves at the location of the discontinuity. If the discontinuity is the first discontinuity, then the incident and reflected waves at its location are the incident and reflected waves at the TDR port and the reflection coefficient can be calculated by dividing by the amplitude of the pulse stimulus. If the discontinuity is preceded by other discontinuities, however, then the incident and reflected waves at its location are not the incident and reflected waves at the TDR port and the peeling algorithm must be used to calculate the reflection coefficient. The peeling algorithm is discussed in section 2.6.1. It should also be noted that although the amplitude and width of the reflection coefficient are dependent on the rise time of the pulse stimulus, the integral of the reflection coefficient is relatively independent of the rise time of the pulse stimulus. A detailed discussion on approximating the excess reactance of inductive and capacitive impedance discontinuities can be found in [5].

$$L \approx 2Z\int_{t1}^{t2} \rho \, dt \tag{11}$$

$$C \approx -\frac{2}{Z}\int_{t1}^{t2} \rho \, dt \tag{12}$$

2.7. PERFORMANCE

The overall performance of the TDR was evaluated based on its performance in four categories: noise, dynamic range, spatial resolution, and acquisition rate. The performance of the TDR in these categories is discussed in the following subsections.

2.7.1. Noise. The dynamic range and spatial resolution of the TDR are limited by noise. The noise is a function of the number of averages, which is one less than the number of acquisitions averaged. If de-embedding is performed, then the noise is also a function of the de-embedded rise time. The TDR was used to measure its own noise by performing a calibration and measuring the 50 Ω termination used to perform the

calibration. The noise was measured with and without de-embedding. The RMS voltage of the noise without de-embedding is plotted in Figure 2.45 as a function of the number of averages. The RMS voltage of the noise with de-embedding is plotted in Figure 2.47 as a function of the de-embedded rise time for 3, 15, and 63 averages. The RMS voltage of the noise with de-embedding is plotted in Figure 2.49 as a function of the de-embedded rise time and the number of averages. The RMS voltage of the noise can be converted to RMS reflection coefficient by dividing by the amplitude of the impulse stimulus. The RMS reflection coefficient of the noise is approximately equal to the RMS voltage of the noise because the amplitude of the impulse stimulus is approximately one volt. The RMS reflection coefficient of the noise without de-embedding is plotted in Figure 2.46 as a function of the number of averages. The RMS reflection coefficient of the noise with de-embedding is plotted in Figure 2.48 as a function of the de-embedded rise time for 3, 15, and 63 averages. The RMS reflection coefficient of the noise with de-embedding is plotted in Figure 2.50 as a function of the de-embedded rise time and the number of averages. From Figure 2.45 through Figure 2.48 it can be seen that averaging attenuates the RMS noise by the square root of the number of acquisitions averaged. This indicates that the noise is primarily Gaussian and uncorrelated to the acquisition. From Figure 2.47 and Figure 2.48 it can be seen that decreasing the de-embedded rise time increases the RMS noise. The de-embedded rise time is inversely proportional to the bandwidth of the Gaussian filter used to filter the de-embedded frequency response of the DUT. Decreasing the de-embedded rise time increases the bandwidth of the Gaussian filter, which increases the bandwidth and RMS noise of the de-embedded frequency response of the DUT. From Figure 2.47 and Figure 2.48 it can also be seen that the slope of the RMS noise above 85 ps increases gradually, while the slope of the RMS noise below 85 ps increases rapidly. This is due to an interaction of the Gaussian filter with the inverse filter used to de-embed the frequency response of the DUT. The inverse filter is the reciprocal of the frequency response of the TDR. The frequency dependent loss in the frequency response of the TDR is converted to frequency dependent gain in the inverse filter. Above 85 ps the loss of the Gaussian filter overcomes the gain of the inverse filter and the slope of the RMS noise increases

gradually. Below 85 ps the gain of the inverse filter overcomes the loss of the Gaussian filter and the slope of the RMS noise increases rapidly.

2.7.2. Dynamic Range. The dynamic range of a TDR is the range of reflection coefficient magnitudes which it can measure. The dynamic range is limited by noise since the maximum reflection coefficient magnitude is one. The dynamic range R_D can be calculated from the RMS reflection coefficient of the noise Γ_N using equation 13. The dynamic range is a function of the de-embedded rise time and the number of averages since the noise is a function of the de-embedded rise time and the number of averages. The dynamic range without de-embedding is plotted in Figure 2.51 as a function of the number of averages. The dynamic range with de-embedding is plotted in Figure 2.52 as a function of the de-embedded rise time for 3, 15, and 63 averages. The dynamic range with de-embedding is plotted in Figure 2.53 as a function of the de-embedded rise time and the number of averages. From Figure 2.51 and Figure 2.52 it can be seen that quadrupling the number of acquisitions averaged increases the dynamic range by 6 dB. From Figure 2.52 it can be seen that the slope of the dynamic range above 85 ps decreases gradually, while the slope of the dynamic range below 85 ps decreases rapidly. This is due to an interaction of the Gaussian filter with the inverse filter used to de-embed the frequency response of the DUT. The interaction of the Gaussian and inverse filters is discussed in section 2.7.1.

$$R_D = 20\log\left(\frac{1}{\Gamma_N}\right) \tag{13}$$

2.7.3. Spatial Resolution. The spatial resolution of a TDR is a measure of its ability to resolve closely spaced impedance discontinuities. The spatial resolution is the minimum separation at which two impedance discontinuities are distinguishable. In other words, the spatial resolution is the separation at which two impedance discontinuities begin to appear as one impedance discontinuity. The spatial resolution in a specified transmission medium is typically defined as half the length of the step pulse edge [6]. The spatial resolution can be calculated from the rise time t_r and propagation velocity v_p using equation 14. The propagation velocity v_p can be measured or calculated

from the effective dielectric constant ε_{eff} of the transmission medium and the speed of light in free space c using equation 15. The spatial resolution as a function of rise time is plotted in Figure 2.54 for effective dielectric constants of 1, 2, 3, 4, and 5. The spatial resolution of the TDR was evaluated on a 50 Ω coaxial cable with four apertures in its outer conductor. The separation of the first and second apertures was 1 cm. The separation of the second and third apertures was 1.5 cm. The separation of the third and fourth apertures was 2 cm. The 50 Ω coaxial cable had a Teflon dielectric with a dielectric constant of approximately 2.1. The measured step response of the 50 Ω coaxial cable with apertures is shown in Figure 2.55 for de-embedded rise times of 50 ps, 100 ps, 150 ps, and 200 ps. From Figure 2.54 it can be seen that the spatial resolution in Teflon for a de-embedded rise time of 100 ps is approximately 1 cm. The first and second apertures in the 50 Ω coaxial cable should be distinguishable for a de-embedded rise time of 100 ps since their separation was 1 cm. From Figure 2.55 it can be seen that the first and second apertures are distinguishable for a de-embedded rise time of 100 ps. From Figure 2.55 it can also be seen that the first and second apertures are indistinguishable for de-embedded rise times of 150 ps and 200 ps. The minimum de-embedded rise time of the TDR is limited by its noise. The noise of the TDR is discussed in section 2.7.1. Decreasing the de-embedded rise time amplifies the noise in the de-embedded impulse and step responses. In order to specify a minimum de-embedded rise time, a maximum noise gain must be specified. The noise gain with de-embedding is plotted in Figure 2.56 as a function of the de-embedded rise time for 3, 15, and 63 averages. From Figure 2.56 it can be seen that the noise gain is not a function of the number of averages. The system rise time can be specified by the 0 dB rise time which is the de-embedded rise time for which the RMS voltage of the noise is equal to the RMS voltage of the noise without de-embedding. From Figure 2.56 it can be seen that the 0 dB rise time is 85 ps which corresponds to a spatial resolution of 9 mm in Teflon. The minimum de-embedded rise time can be specified by the 6 dB rise time which is the de-embedded rise time for which the RMS voltage of the noise is twice the RMS voltage of the noise without de-embedding. From Figure 2.56 it can be seen that the 6 dB rise time is 60 ps which corresponds to a spatial resolution of 6 mm in Teflon.

$$l = \frac{1}{2} t_r v_p \tag{14}$$

$$v_p = \frac{c}{\sqrt{\varepsilon_{eff}}} \tag{15}$$

2.7.4. Acquisition Rate. The acquisition rate of the TDR was evaluated with the HMC346 voltage variable attenuator shown in Figure 2.57. The output of the attenuator was shorted to create a reflection coefficient which was inversely proportional to the attenuation. The attenuator was then modulated with a signal generator to create a time varying reflection coefficient which was a function of the modulation signal. The concept is illustrated in Figure 2.58. The attenuator was modulated with different modulation signals and the impulse response was monitored by the TDR. The impulse response for 1 kHz sine modulation is shown in Figure 2.59. The impulse response was acquired at the maximum acquisition frequency of 10.1725 kHz. The modulated reflected pulse appears as a row of dots. The 1 ms separation of the dots corresponds to the 1 kHz modulation frequency. The impulse response for 100 Hz sine modulation is shown in Figure 2.60. The impulse response was acquired at an acquisition frequency of 1.27157 kHz. The modulated reflected pulse appears as a row of dots. The 10 ms separation of the dots corresponds to the 100 Hz modulation frequency. The impulse response for 100 Hz square modulation is shown in Figure 2.61. The impulse response was acquired at an acquisition frequency of 1.27157 kHz. The modulated reflected pulse appears as a row of dashes. The 10 ms separation of the dashes corresponds to the 100 Hz modulation frequency. The impulse response for 100 Hz ramp modulation is shown in Figure 2.62. The impulse response was acquired at an acquisition frequency of 1.27157 kHz. The modulated reflected pulse appears as a row of smeared dots. The 10 ms separation of the smeared dots corresponds to the 100 Hz modulation frequency. From Figure 2.59 through Figure 2.62 it can be seen that the TDR can monitor time-variant impedance discontinuities with frequency content as high as 1 kHz. The acquisition rate of the TDR was further evaluated with a transmission line formed from a wire suspended above a ground plane. The transmission line is shown in Figure 2.63. One end of the wire was connected to the ground plane, while the other end was

connected to an SMA connector. The wire was connected so that it would vibrate when plucked like a guitar string. The vibration of the wire would modulate the impedance of the transmission line which would create a time varying reflection coefficient. The wire was plucked like a guitar string and the step response of the transmission line was monitored by the TDR. The step response of the transmission line is shown in Figure 2.64. The vibration of the wire appears as two rows of alternating red and blue dots. The separation of the dots indicates a vibration frequency of approximately 117 Hz. The 2.6 ns length of the dots corresponds to the 39 cm length of the transmission line. The two rows of dots indicate a mechanical wavelength of 39 cm. From Figure 2.64 it can be seen that time-variant impedance discontinuities caused by vibration can be monitored so long as their frequency and magnitude are well within the acquisition rate and dynamic range of the TDR.

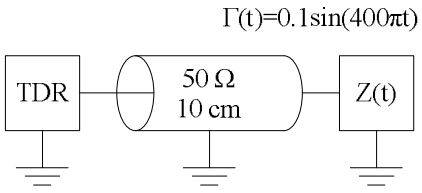

Figure 2.1. Hypothetical Measurement Setup

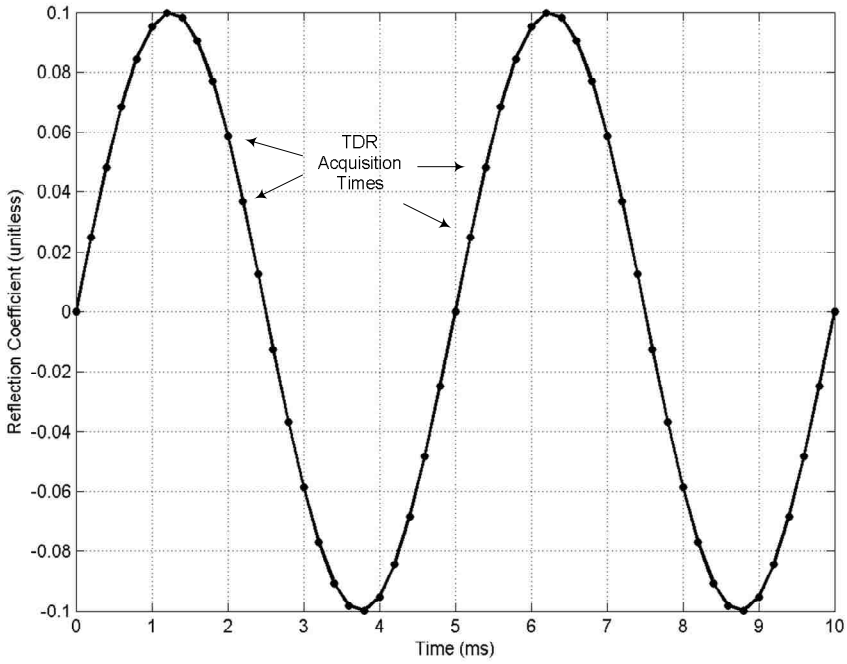

Figure 2.2. Reflection Coefficient as a Function of Time

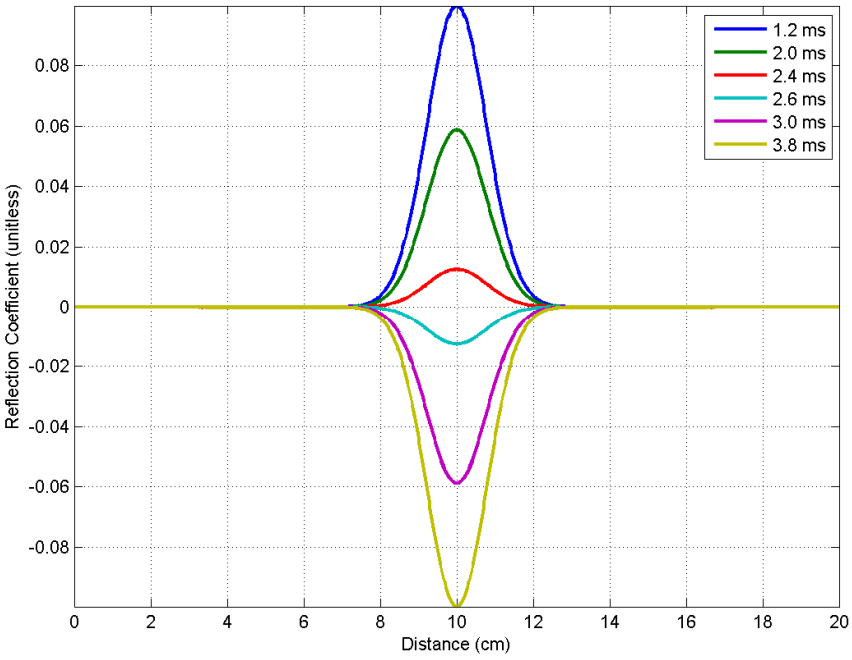

Figure 2.3. TDR Impulse Response at Various Times

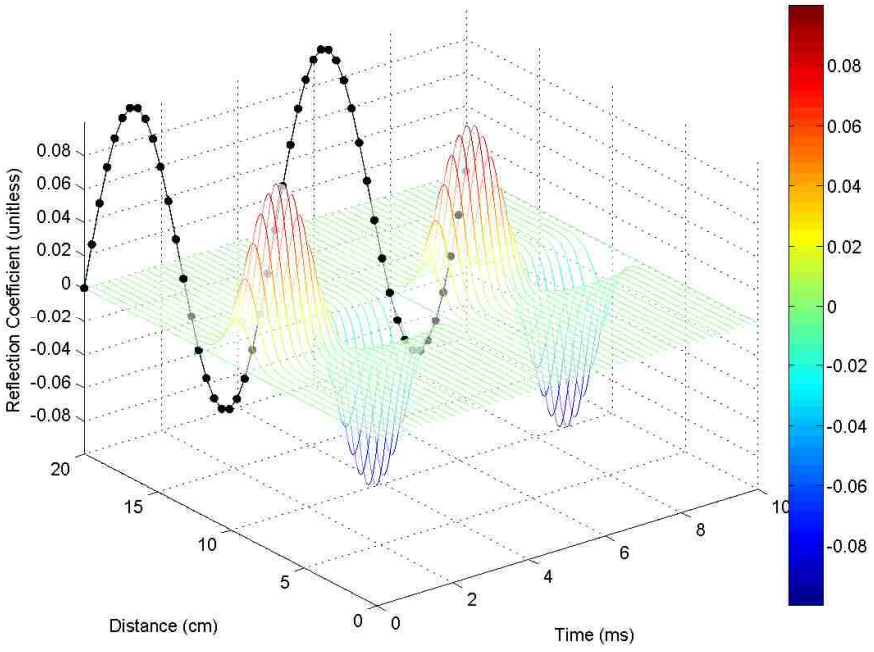

Figure 2.4. TDR Impulse Response as a Function of Time

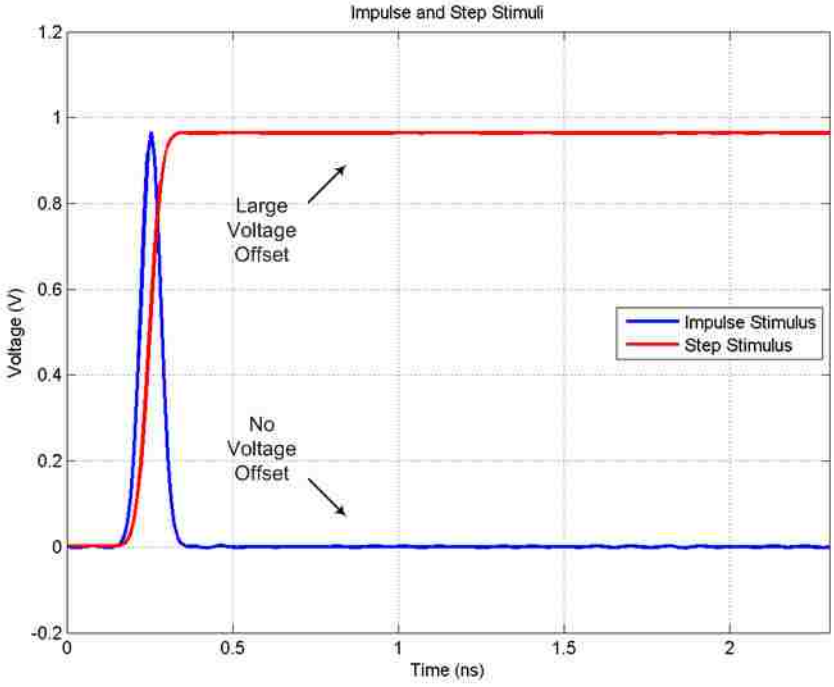

Figure 2.5. Impulse and Step Stimuli

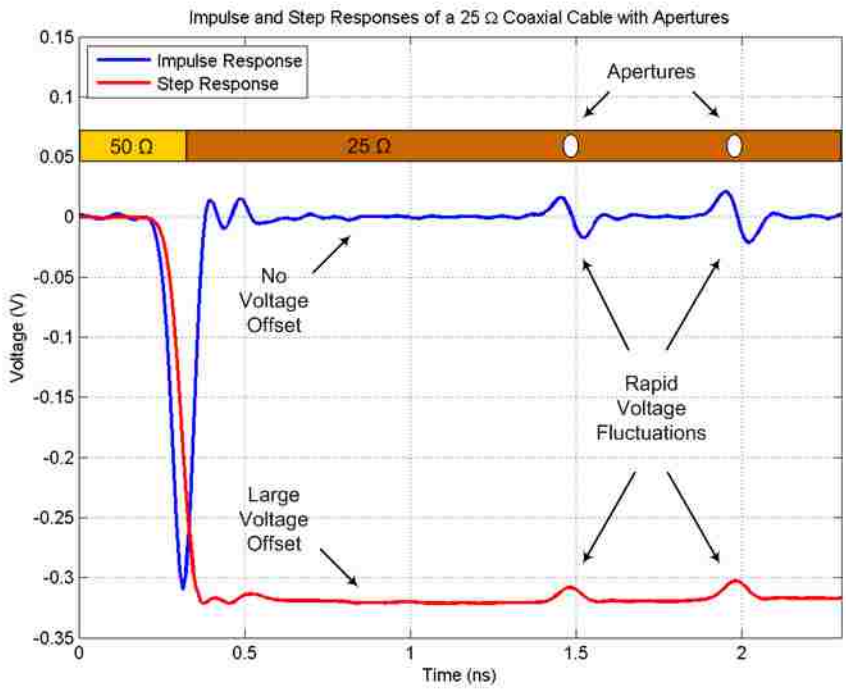

Figure 2.6. Impulse and Step Responses

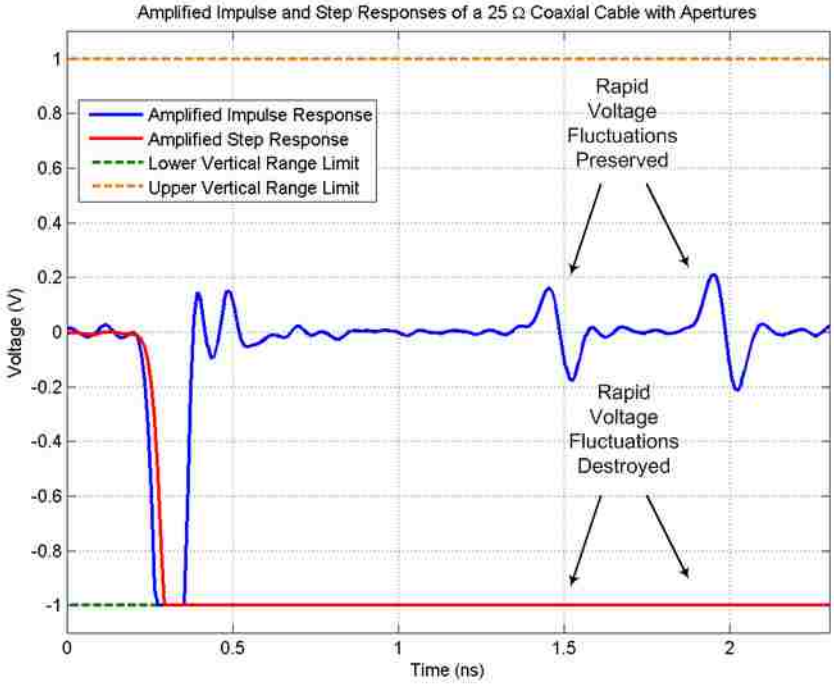

Figure 2.7. Amplified Impulse and Step Responses

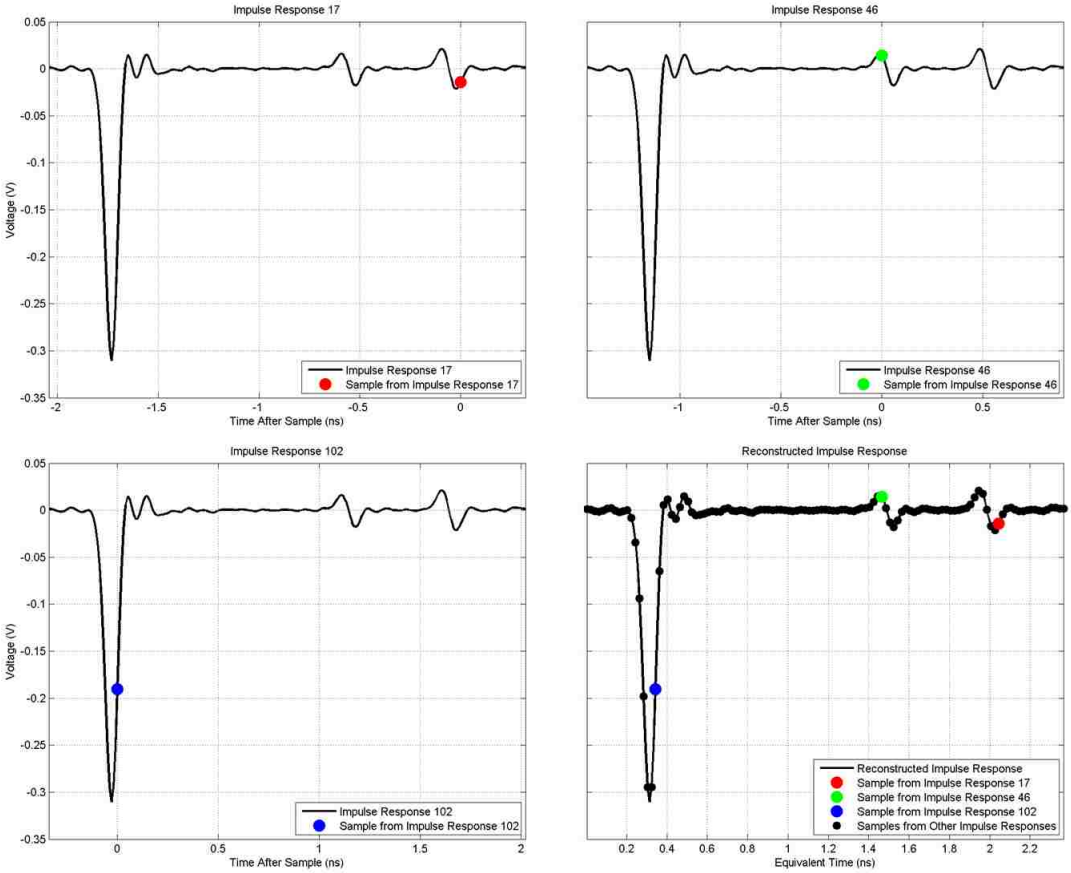

Figure 2.8. Sequential Equivalent-Time Sampling

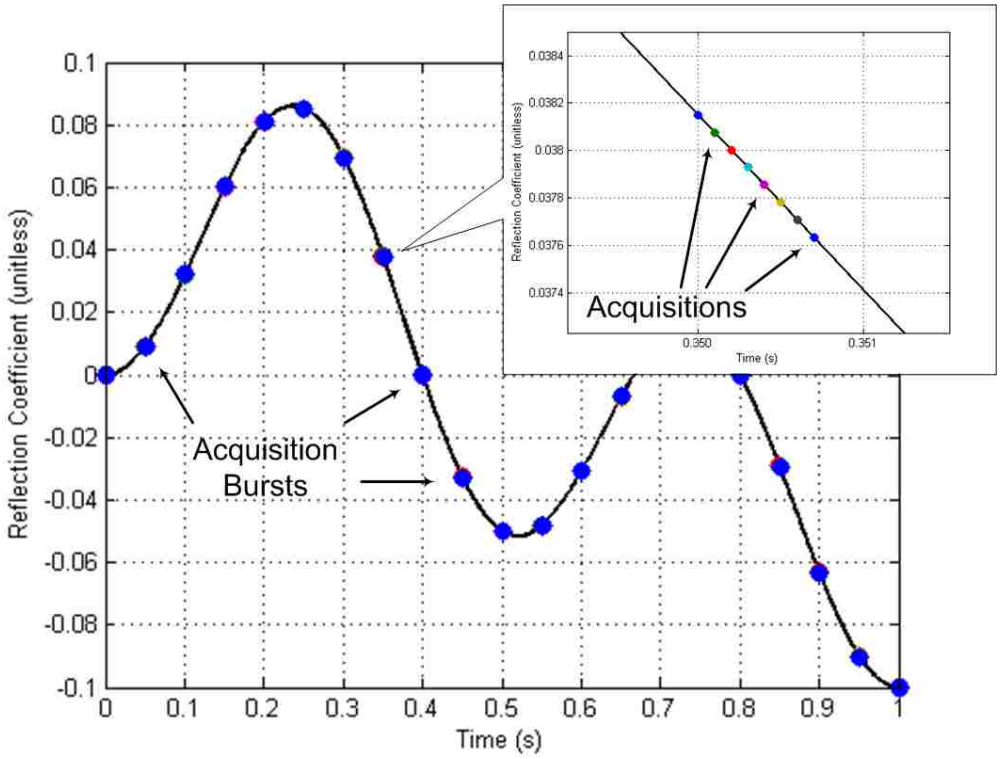

Figure 2.9. Burst Orientation of the Multiple Acquisition Mode

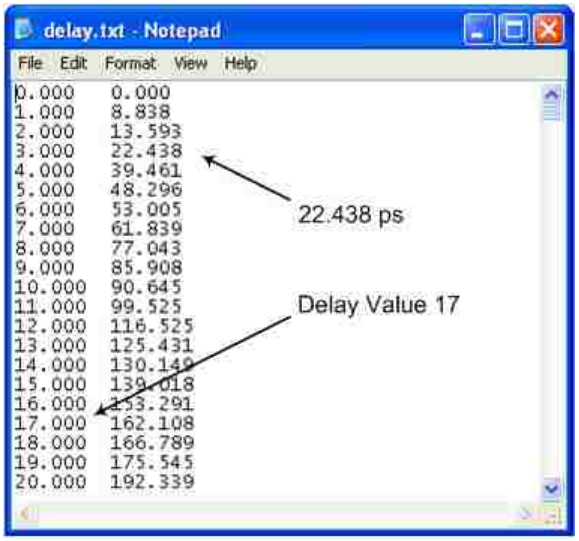

Figure 2.10. Portion of Text File Generated by Horizontal Calibration

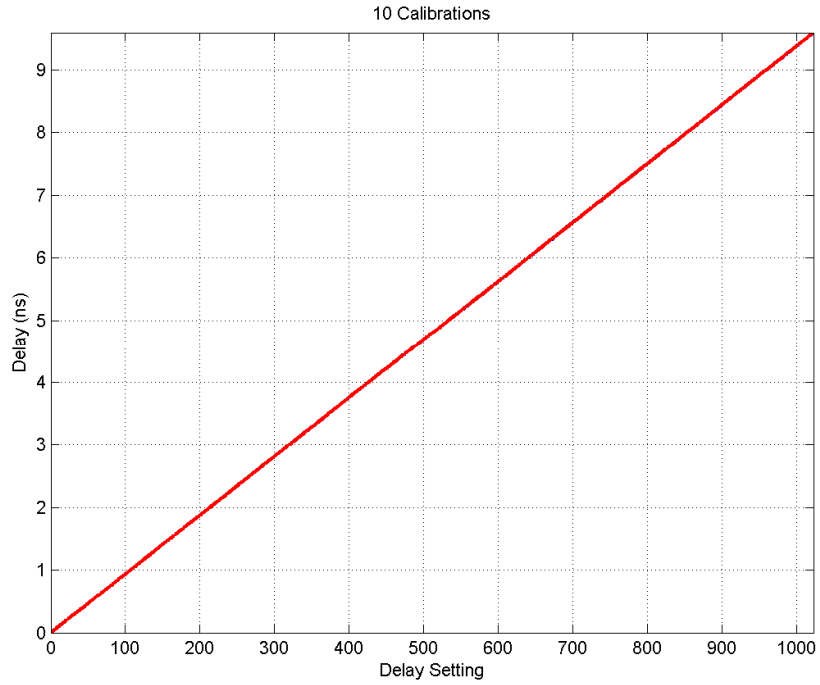

Figure 2.11. Delay Values

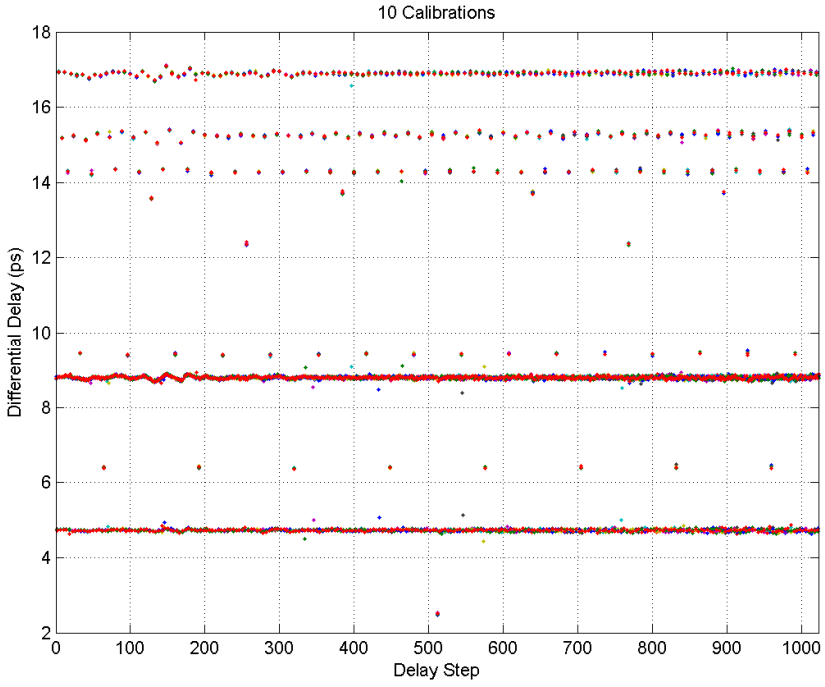

Figure 2.12. Differential Delay Values

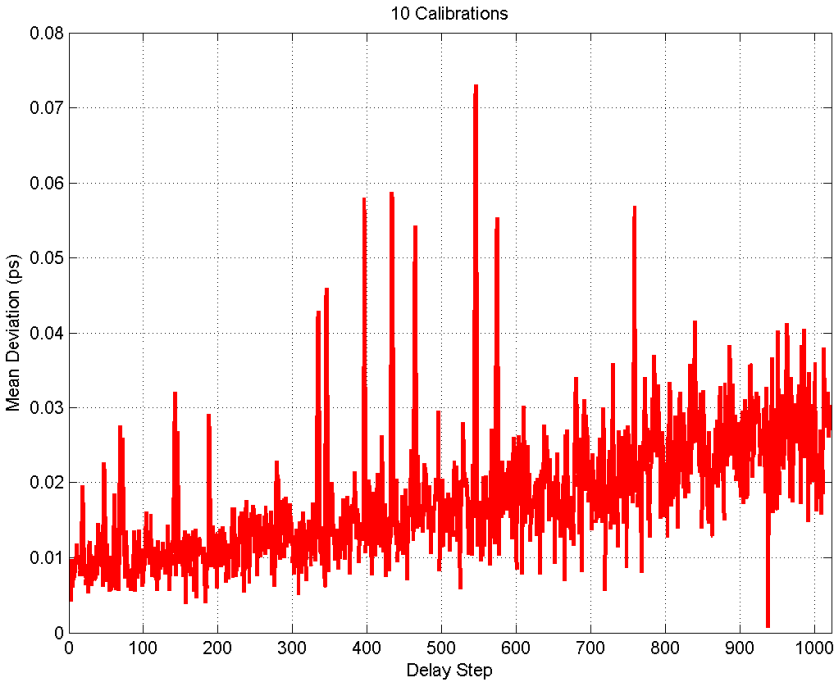

Figure 2.13. Mean Deviations of Differential Delay Values

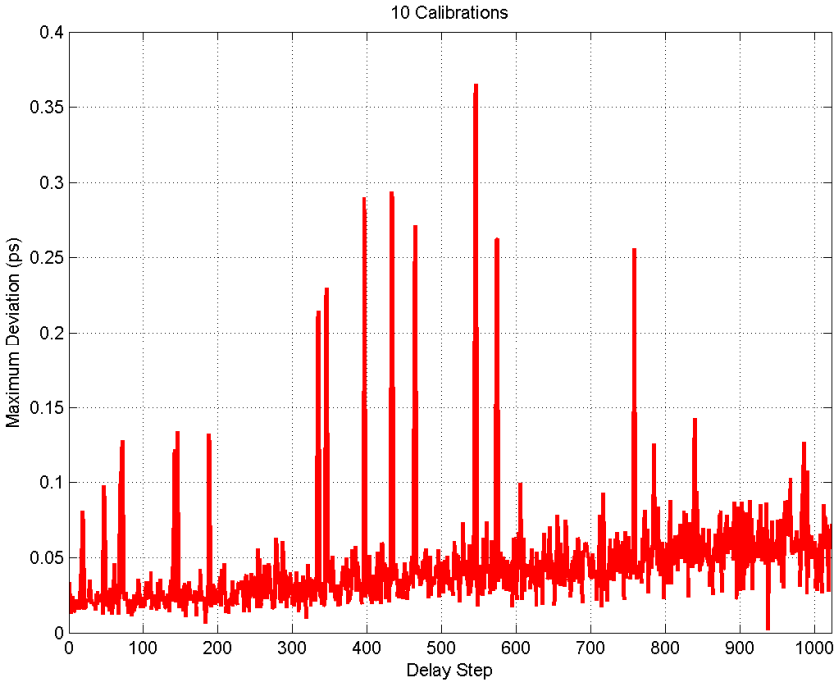

Figure 2.14. Maximum Deviations of Differential Delay Values

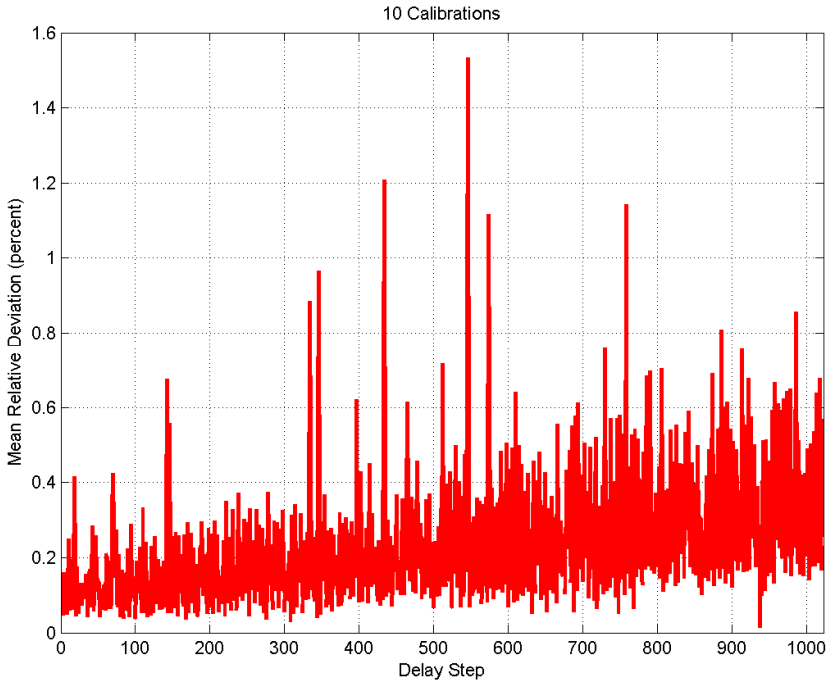

Figure 2.15. Mean Relative Deviations of Differential Delay Values

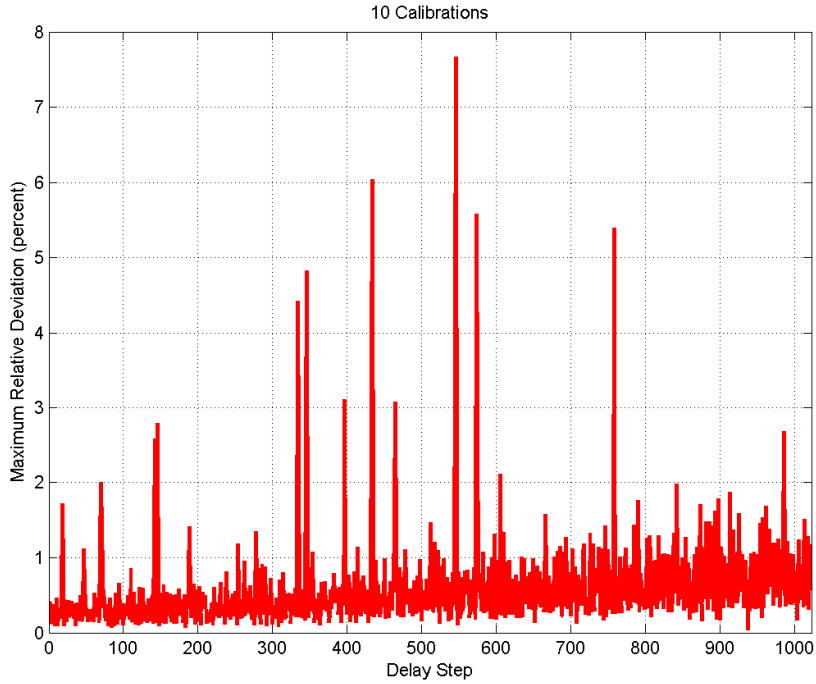

Figure 2.16. Maximum Relative Deviations of Differential Delay Values

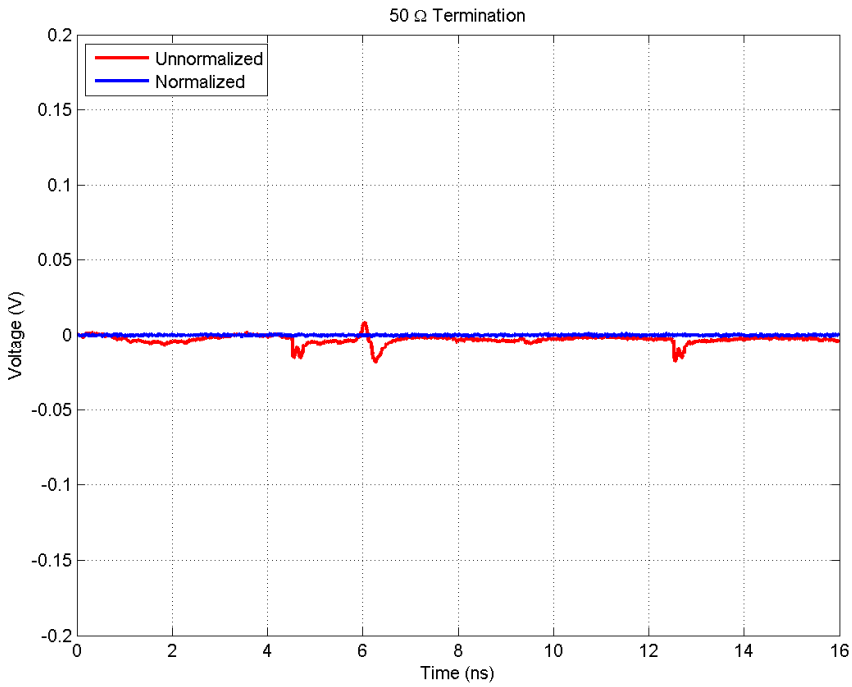

Figure 2.17. Normalization of a 50 Ω Termination

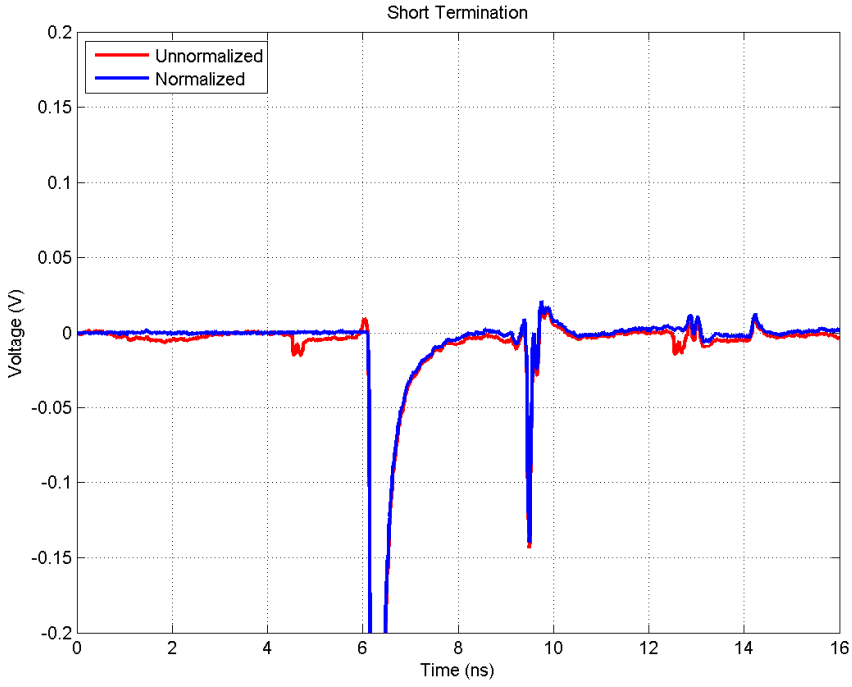

Figure 2.18. Normalization of a Short Termination

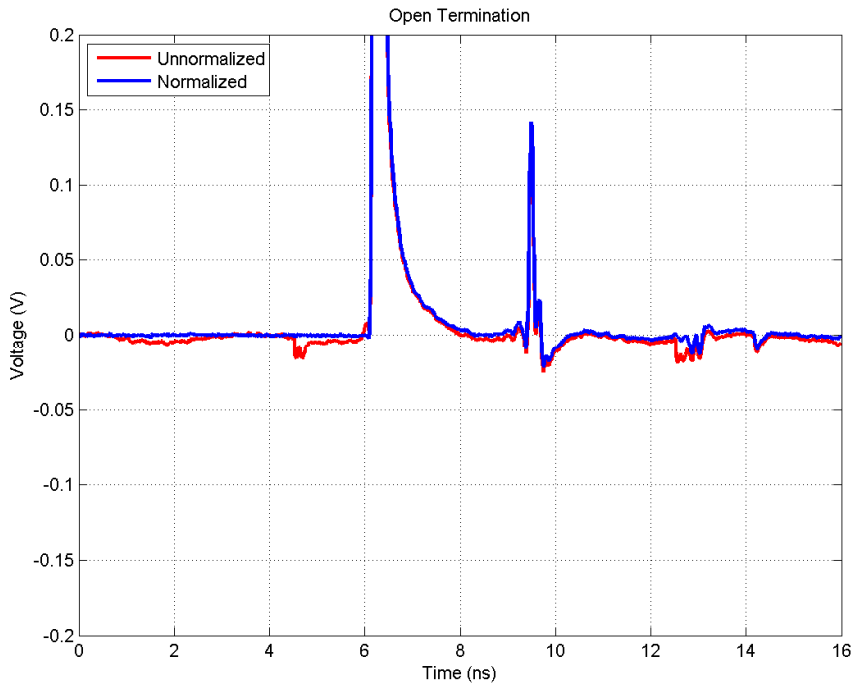

Figure 2.19. Normalization of an Open Termination

```
vi=importdata('dut.txt');
y=importdata('short.txt');
vi=vi.data;
y=y.data;
ti=vi(:,1);
vi=vi(:,2);                                          % embedded impulse response of dut
y=y(:,2);                                            % negative impulse response of tdr
y=-y;                                                % positive impulse response of tdr
tr=100e-12;                                          % rise time of gaussian filter
a=sqrt(2)*erfinv(0.8)*(max(ti)-min(ti))/tr;          % reciprocal of standard deviation
x=max(y)*gausswin(length(y),a);                      % impulse response of gaussian filter
X=fft(x);                                            % frequency response of gaussian filter
Y=fft(y);                                            % frequency response of tdr
H=Y./X;
VI=fft(vi);                                          % embedded frequency response of dut
VO=VI./H;                                            % de-embedded frequency response of dut
fs=1/mean(diff(ti));                                 % sample rate
f=[0:length(VO)-1]'*fs/(length(VO)-1);               % frequency array for de-embedded frequency response of dut
t=ti(y==max(y))-mean(ti);                            % time delay of tdr impulse response
VO=VO.*exp(-1i*2*pi*f*t);                            % delay impulse response of dut
fmax=min([min(f(isnan(VO)==1)),min(f(isinf(VO)==1))]); % check for Inf or NaN values caused by division
if isempty(fmax)                                     % no Inf or NaN values
    fmax=fs/2;                                       % maximum freqeuncy is half sample rate
end
VO=[VO(1);VO(f<fmax & f>0);flipud(conj(VO(f<fmax & f>0)))]; % remove Inf or NaN values caused by division
to=[min(ti):(max(ti)-min(ti))/(length(VO)-1):max(ti)]';  % time array for de-embedded impulse response of dut
vo=ifft(VO);                                         % de-embedded impulse response of dut
```

Figure 2.20. MATLAB Implementation of De-embedding Algorithm

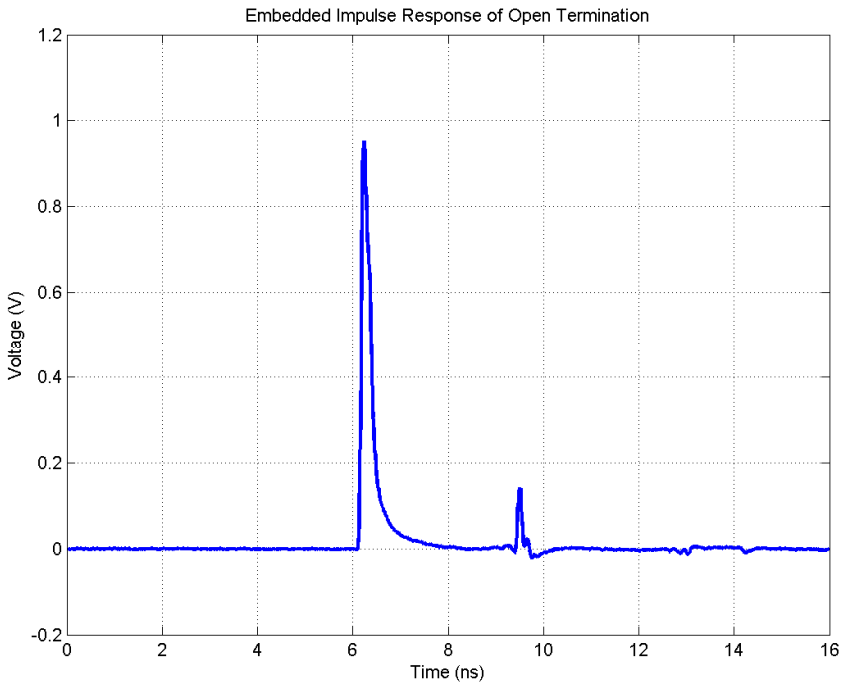

Figure 2.21. Embedded Impulse Response of an Open Termination

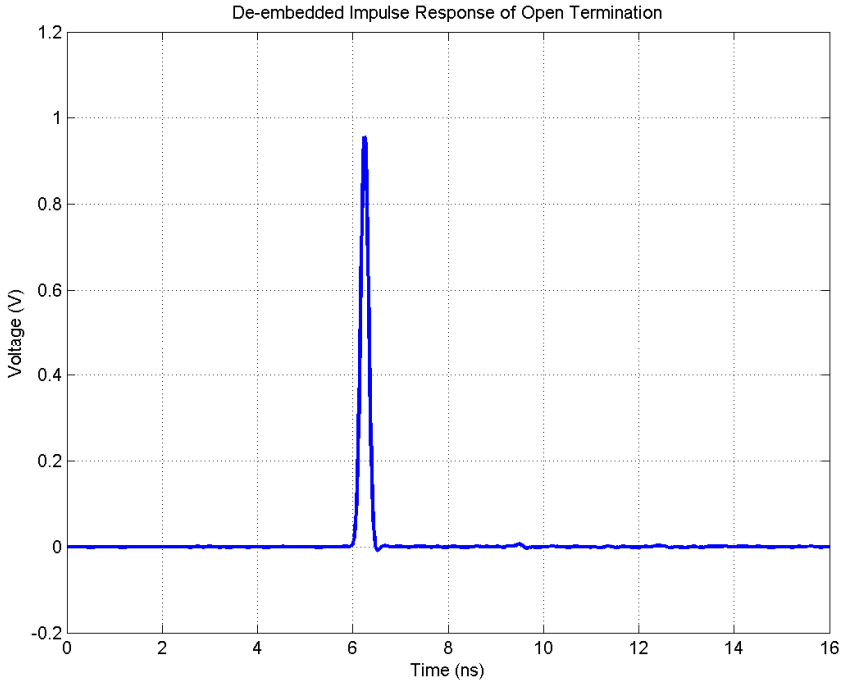

Figure 2.22. De-embedded Impulse Response of an Open Termination

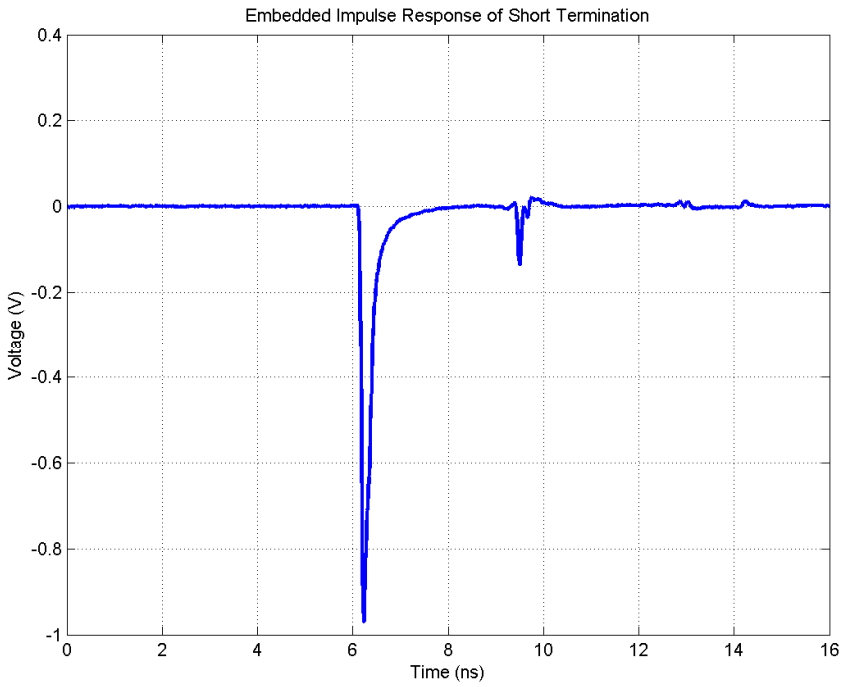

Figure 2.23. Embedded Impulse Response of a Short Termination

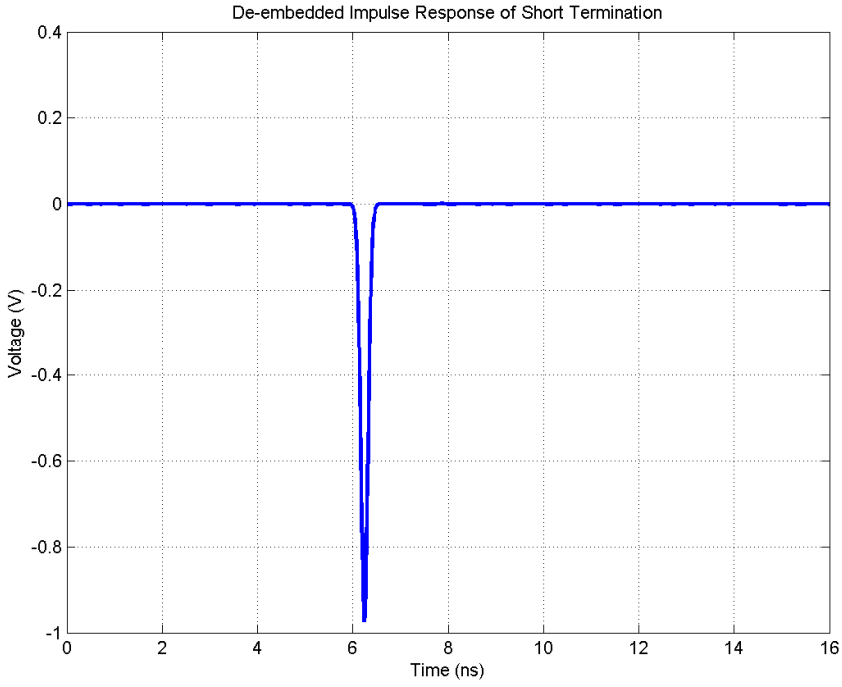

Figure 2.24. De-embedded Impulse Response of a Short Termination

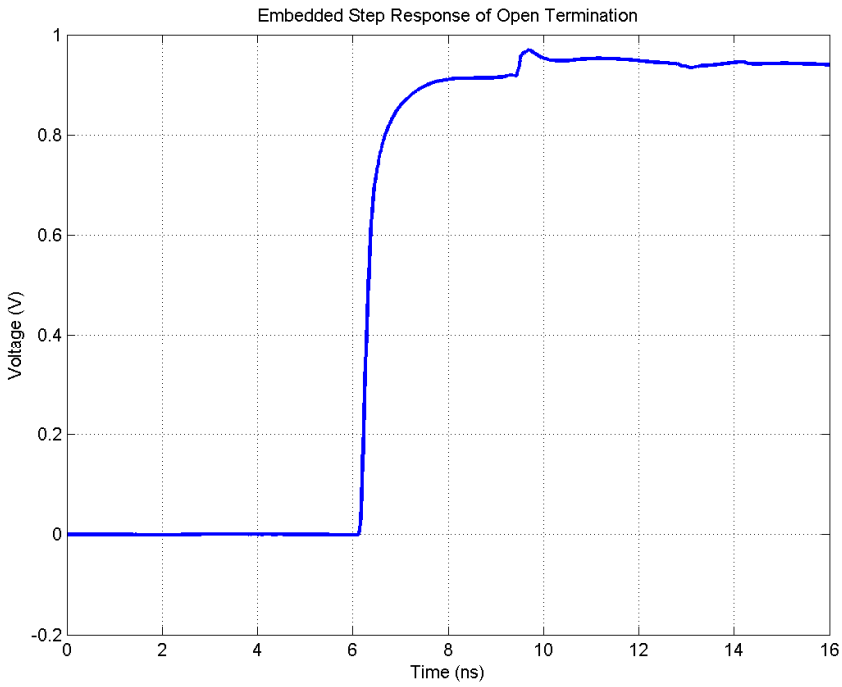

Figure 2.25. Embedded Step Response of an Open Termination

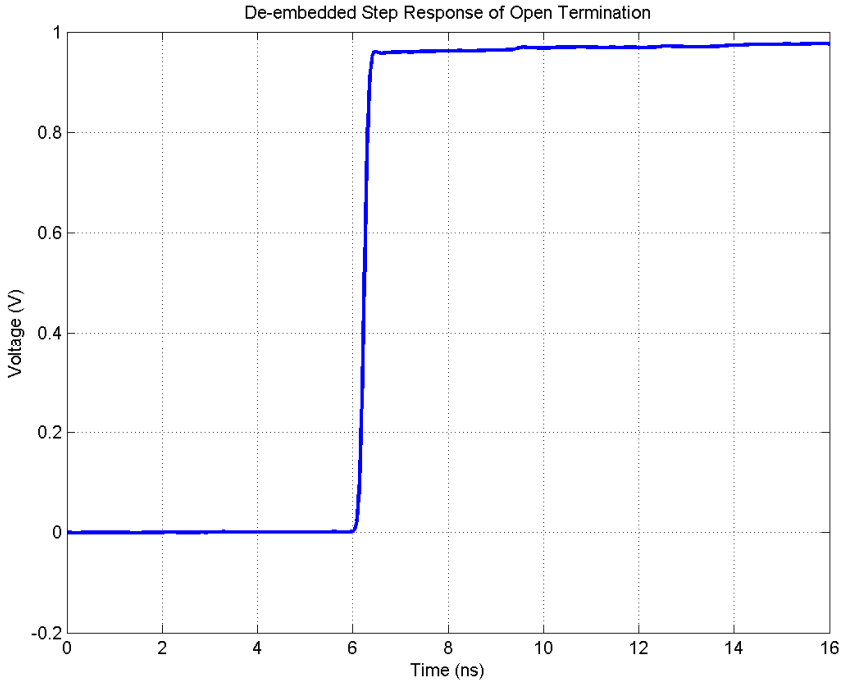

Figure 2.26. De-embedded Step Response of an Open Termination

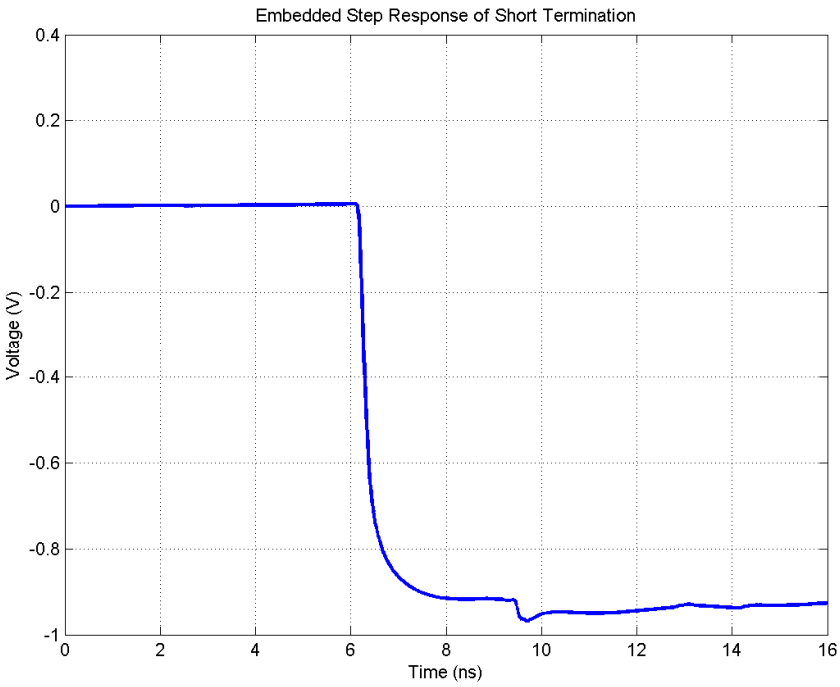

Figure 2.27. Embedded Step Response of a Short Termination

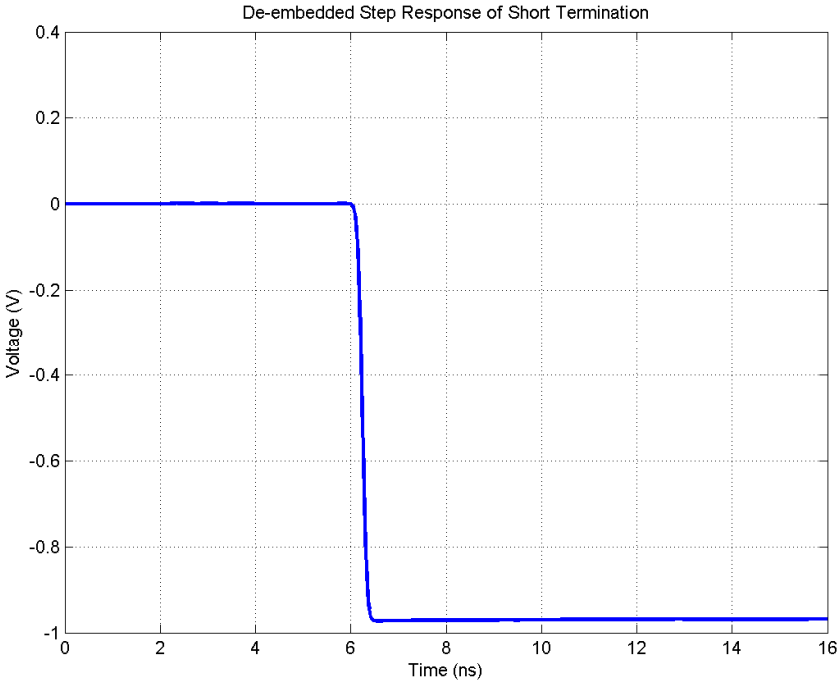

Figure 2.28. De-embedded Step Response of a Short Termination

```
rm=importdata('short.txt');
lm=importdata('dut.txt');
rm=rm.data;
lm=lm.data;
t=rm(:,1);
rm=rm(:,2);
lm=interp1(lm(:,1),lm(:,2),t);                  % sample reflected wave at same times as incident wave
rm=max(abs(rm))*ones(length(rm),1);             % make incident wave constant
n=length(lm);
z(1)=50;                                        % reference impedance
for i=1:n
    k(i)=lm(1)/rm(1);                           % reflection coefficient
    z(i+1)=z(i)*(1+k(i))/(1-k(i));              % impedance
    rp=-k(i)*lm(1:n+1-i)+rm(1:n+1-i);           % positive right going wave
    lp=-k(i)*rm(1:n+1-i)+lm(1:n+1-i);           % positive left going wave
    rm=rp(1:n-i);                               % negative right going wave
    lm=lp(2:n+1-i);                             % negative left going wave
end
z=z(2:n+1);                                     % impedances
z(find(z<=0,1):length(z))=z(find(z<=0,1)-1);    % remove negative values
z(find(isinf(z),1):length(z))=z(find(isinf(z),1)-1);  % remove INF values
k=(z-50)./(z+50);                               % reflection coefficients
```

Figure 2.29. MATLAB Implementation of the Peeling Algorithm

Figure 2.30. Stepped-Impedance Transmission Line

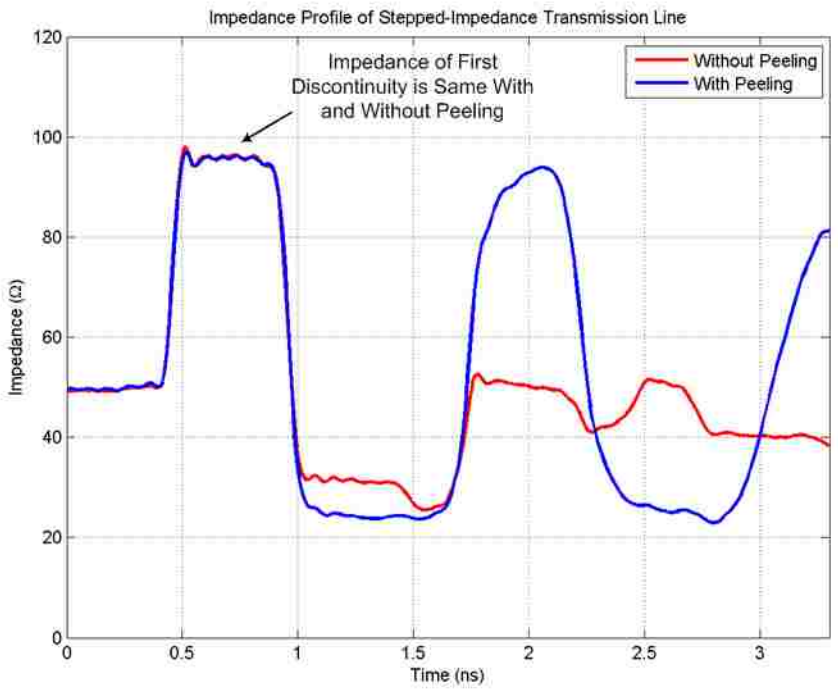

Figure 2.31. Impedance Profile of Stepped-Impedance Transmission Line

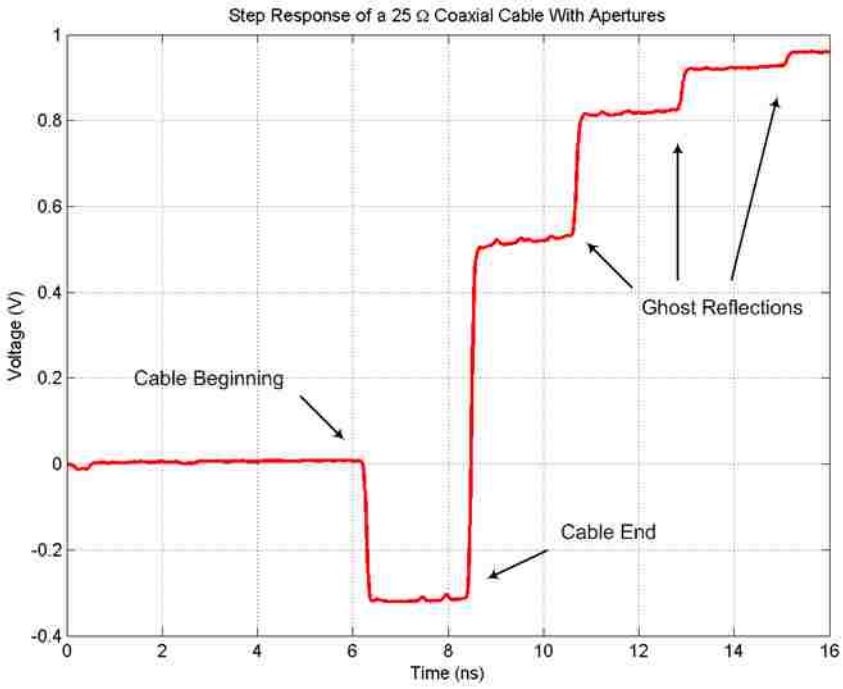

Figure 2.32. Ghost Reflections in Step Response

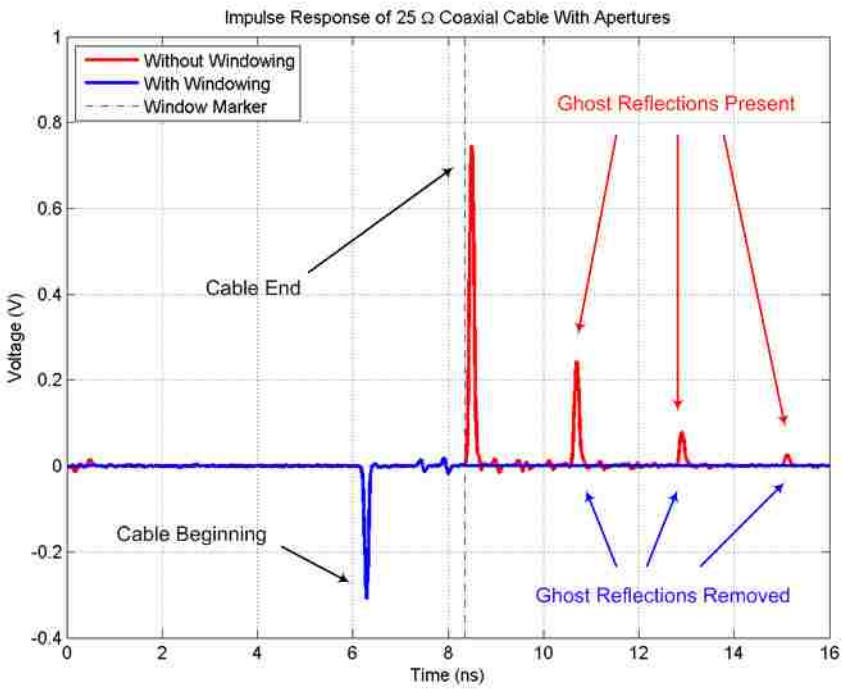

Figure 2.33. Removal of Ghost Reflections in Impulse Response

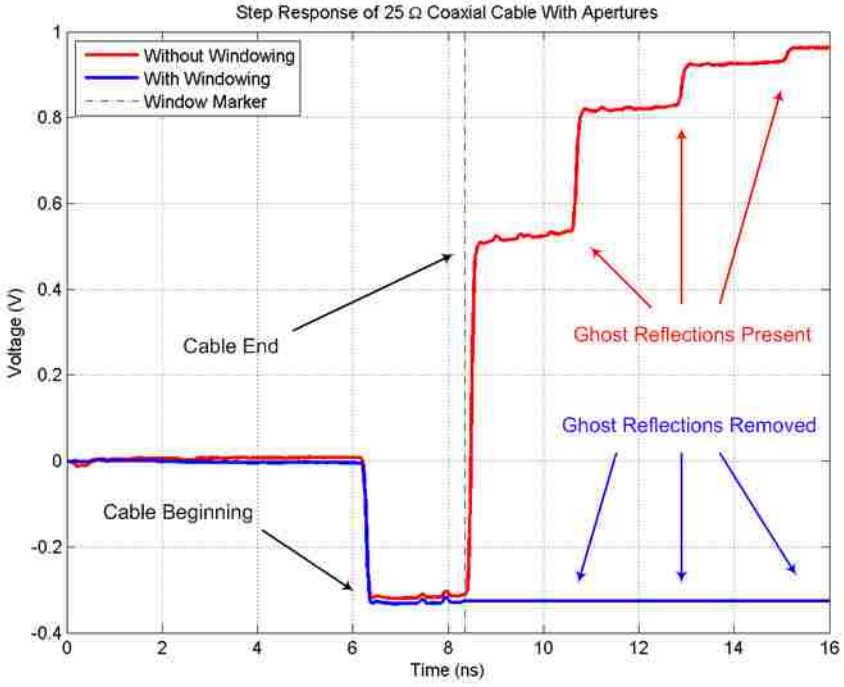

Figure 2.34. Removal of Ghost Reflections in Step Response

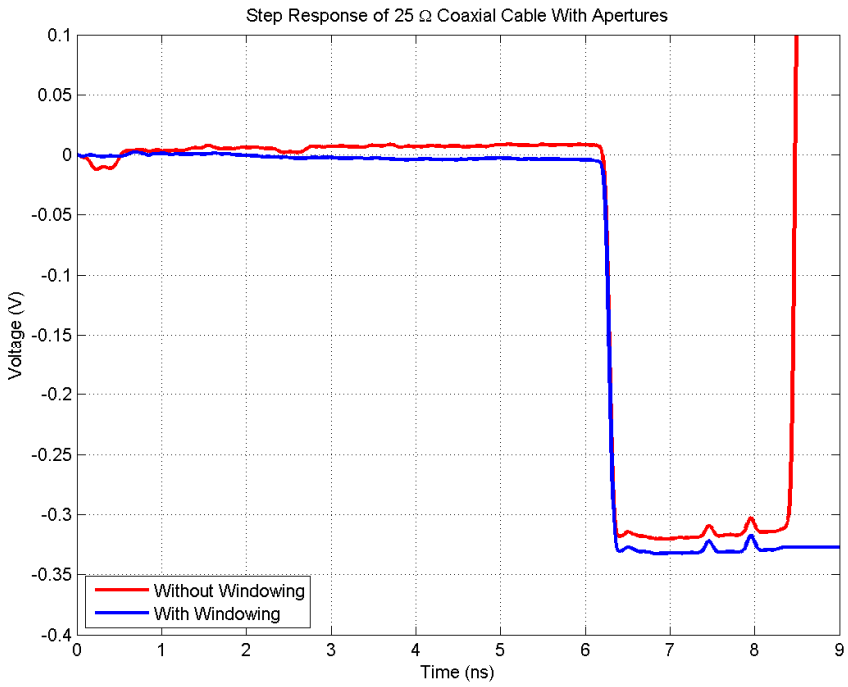

Figure 2.35. Interference of Ghost Reflections with De-embedding

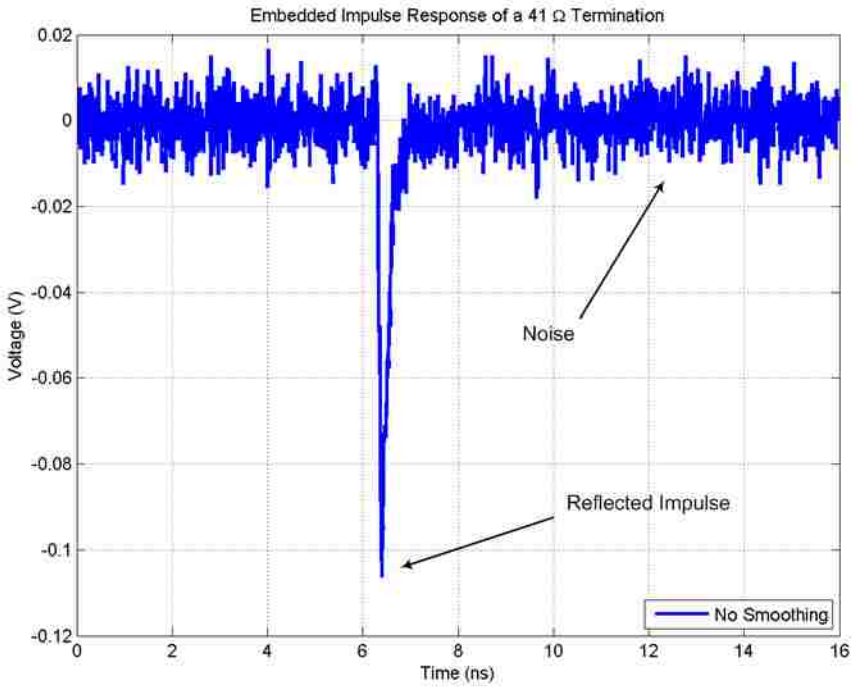

Figure 2.36. Embedded Impulse Response Without Smoothing

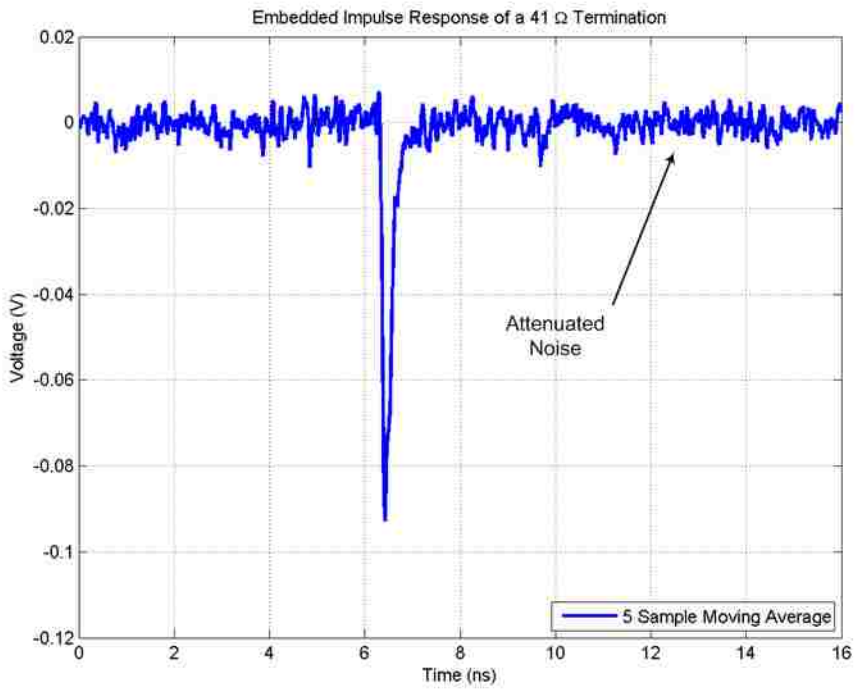

Figure 2.37. Embedded Impulse Response With Smoothing

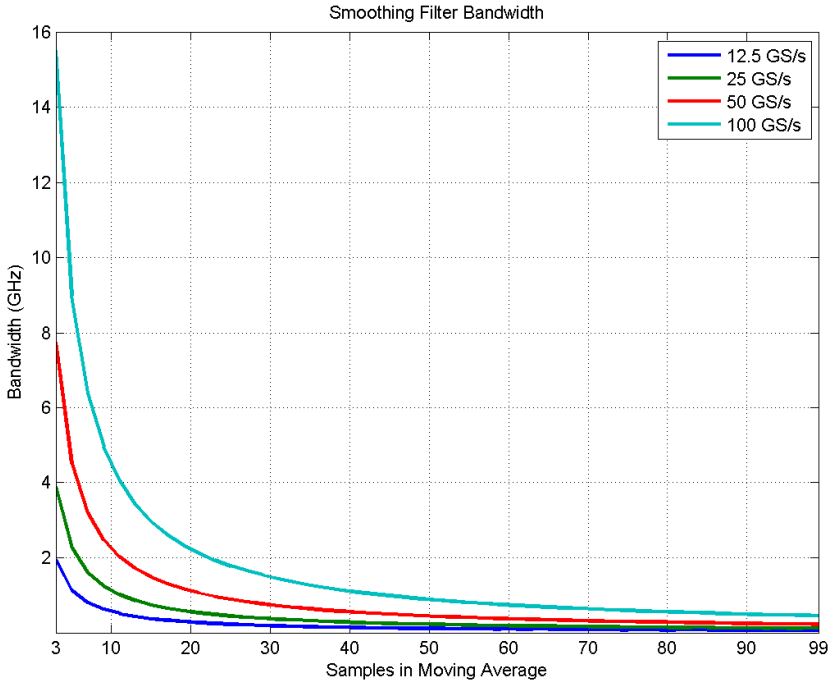

Figure 2.38. Smoothing Filter Bandwidth in GHz

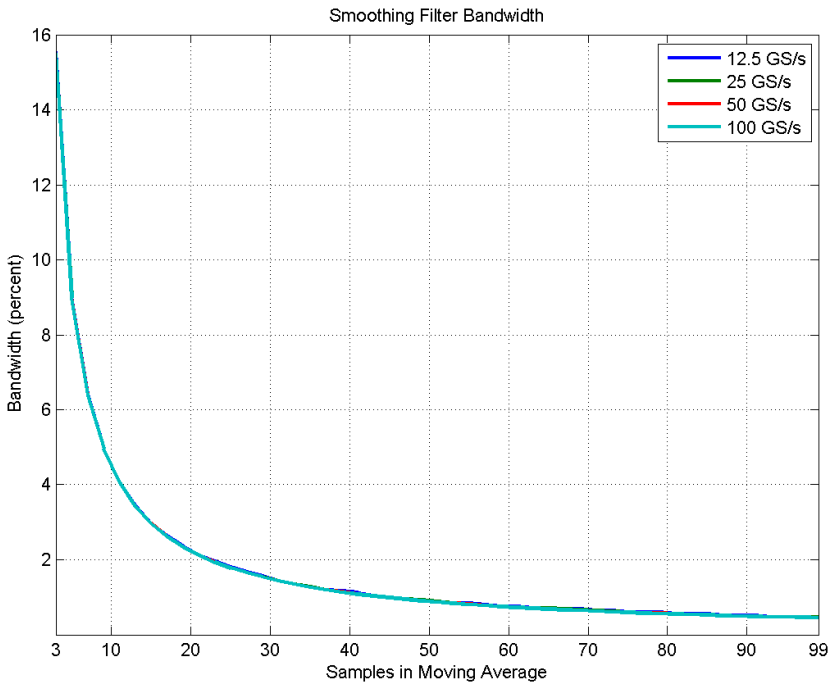

Figure 2.39. Smoothing Filter Bandwidth in Percent

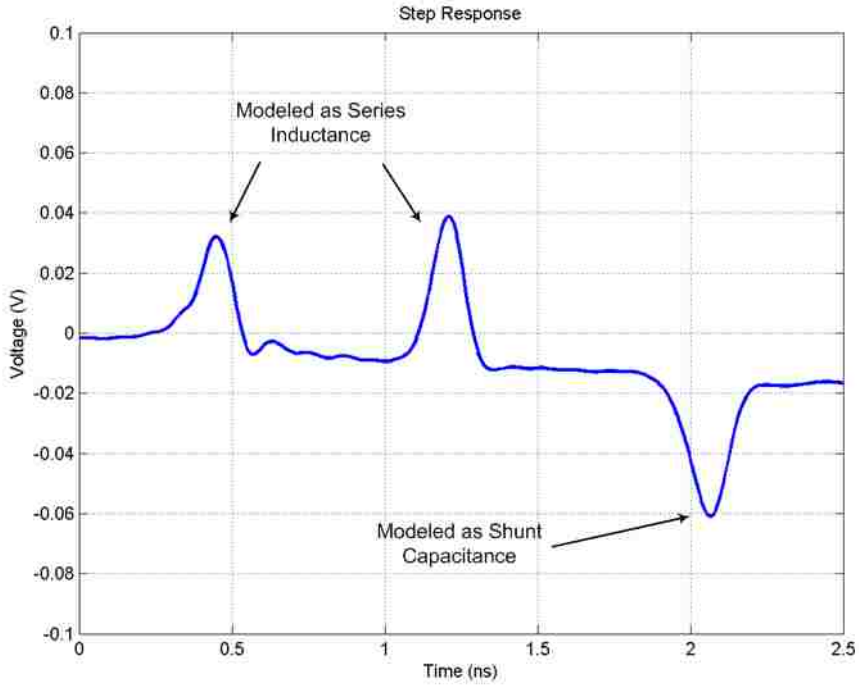

Figure 2.40. Inductive and Capacitive Impedance Discontinuities

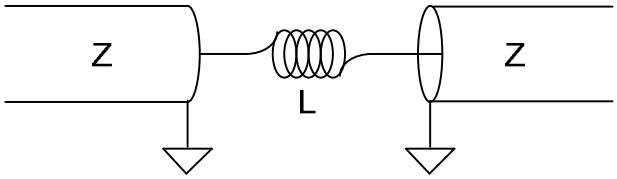

Figure 2.41. Series Inductance Model

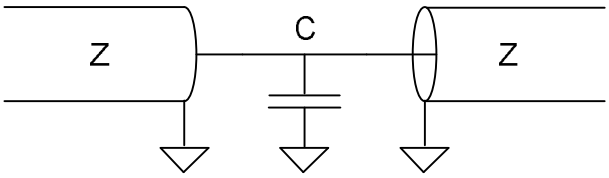

Figure 2.42. Shunt Capacitance Model

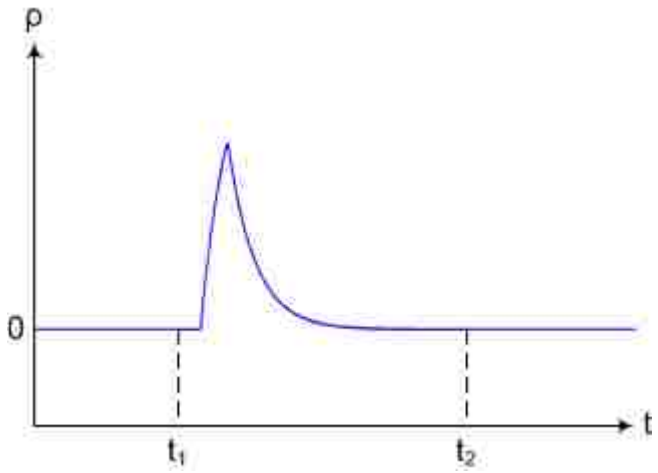

Figure 2.43. Response of the Series Inductance Model

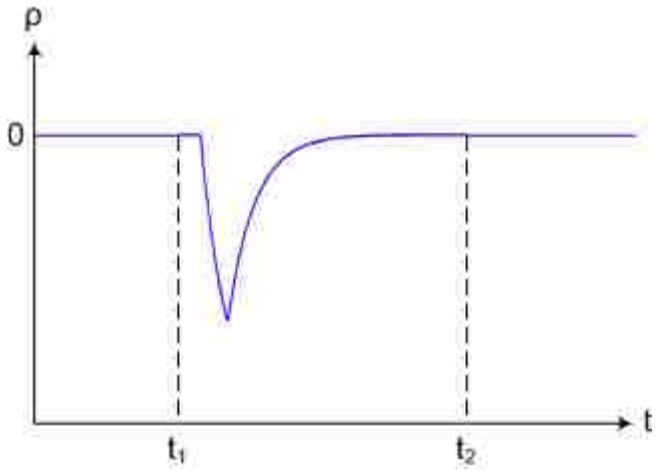

Figure 2.44. Response of the Shunt Capacitance Model

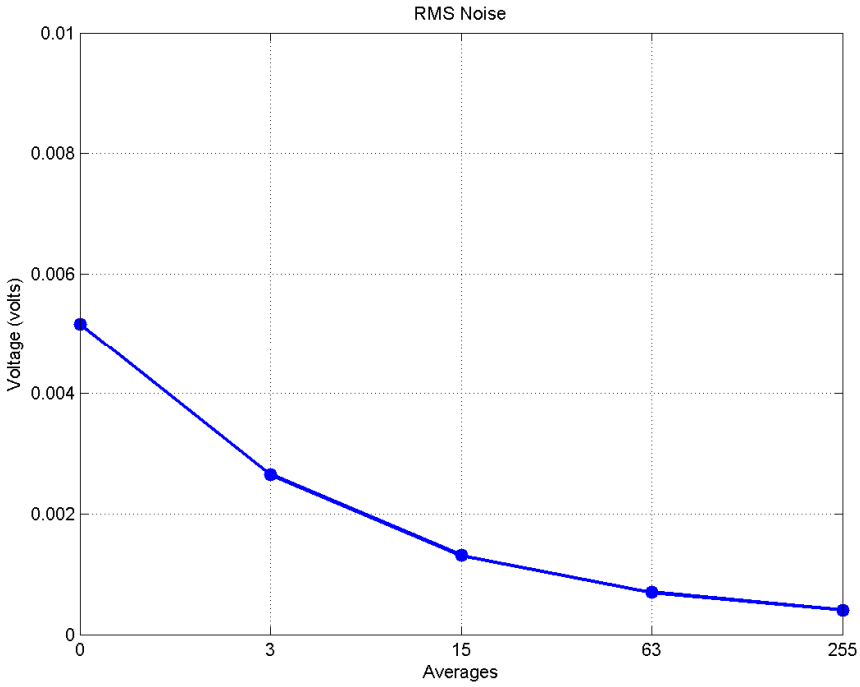

Figure 2.45. RMS Noise Voltage Without De-embedding

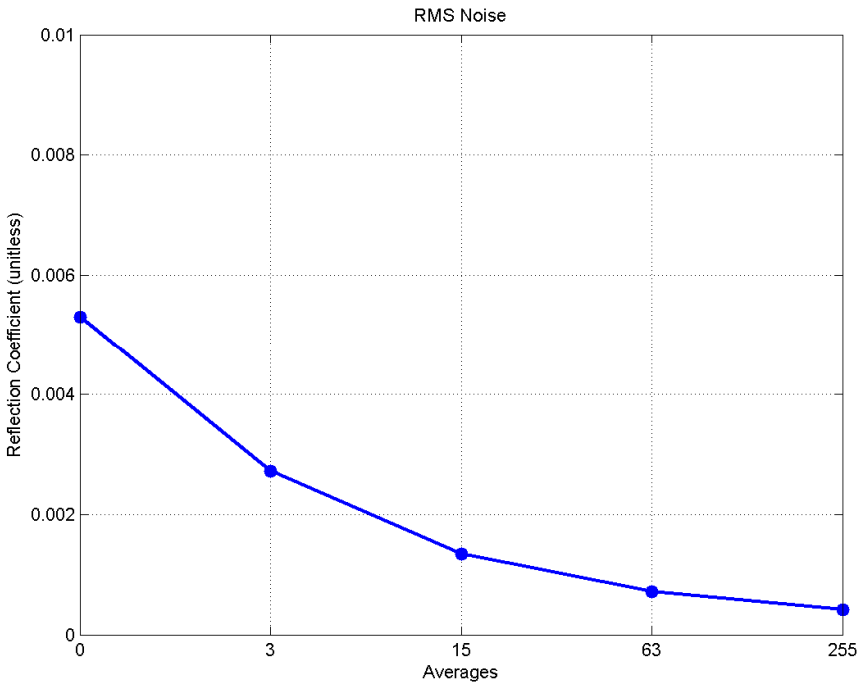

Figure 2.46. RMS Noise Reflection Coefficient Without De-embedding

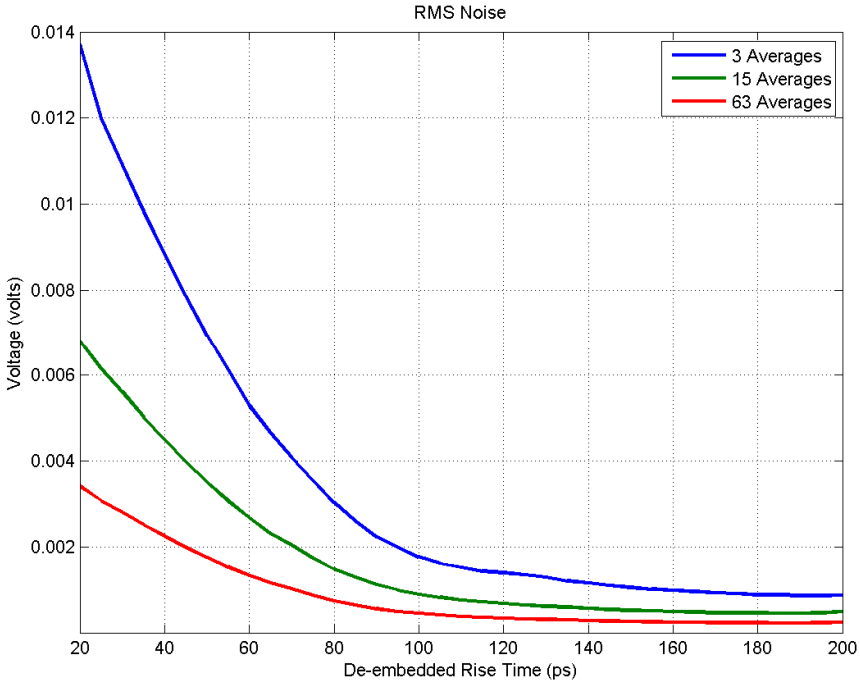

Figure 2.47. RMS Noise Voltage With De-embedding

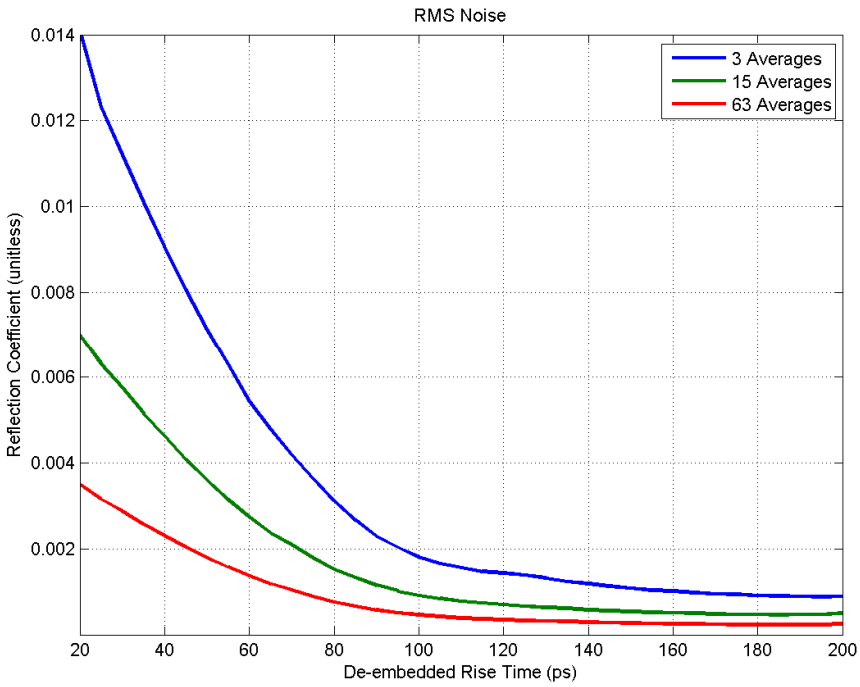

Figure 2.48. RMS Noise Reflection Coefficient With De-embedding

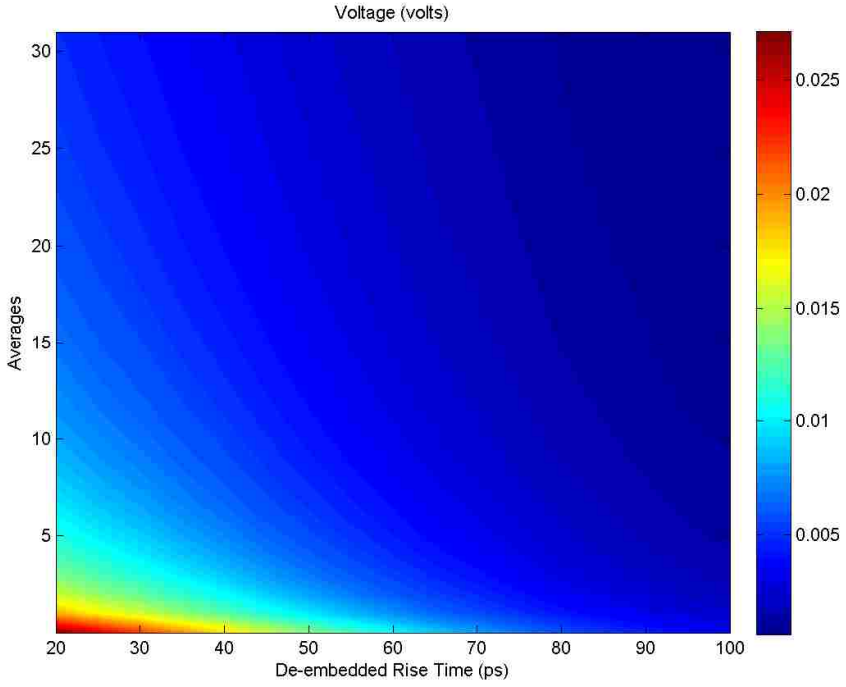

Figure 2.49. RMS Noise Voltage With De-embedding

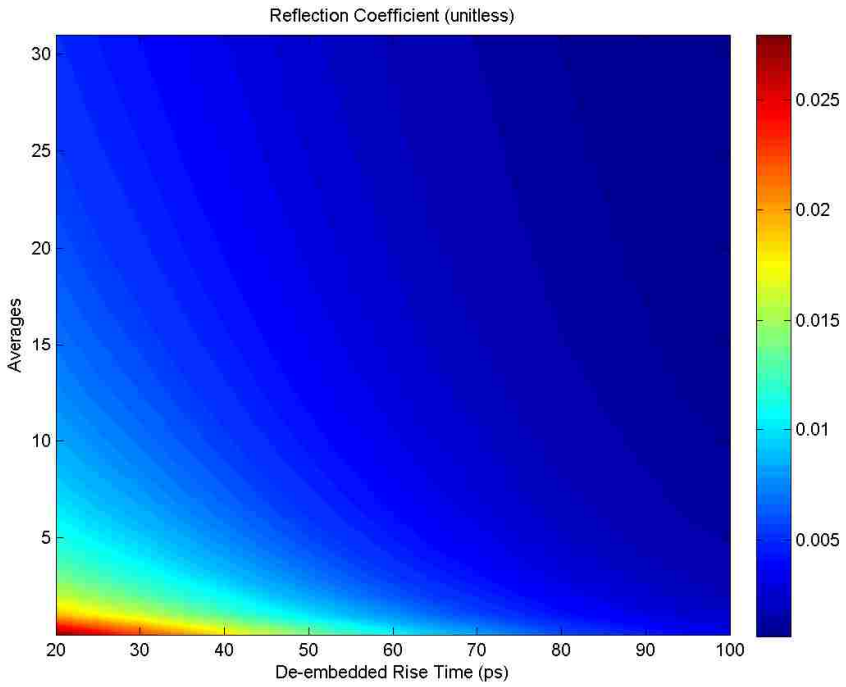

Figure 2.50. RMS Noise Reflection Coefficient With De-embedding

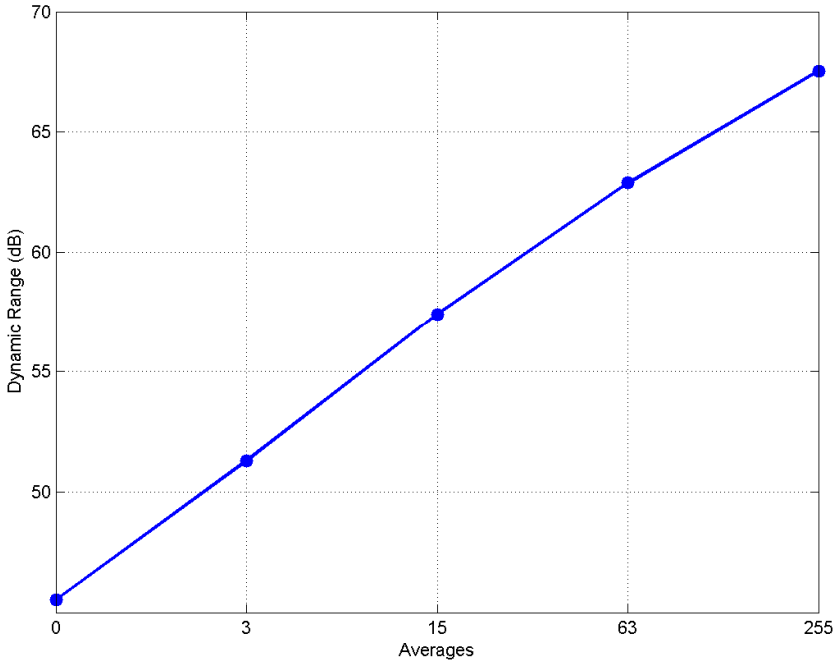

Figure 2.51. Dynamic Range Without De-embedding

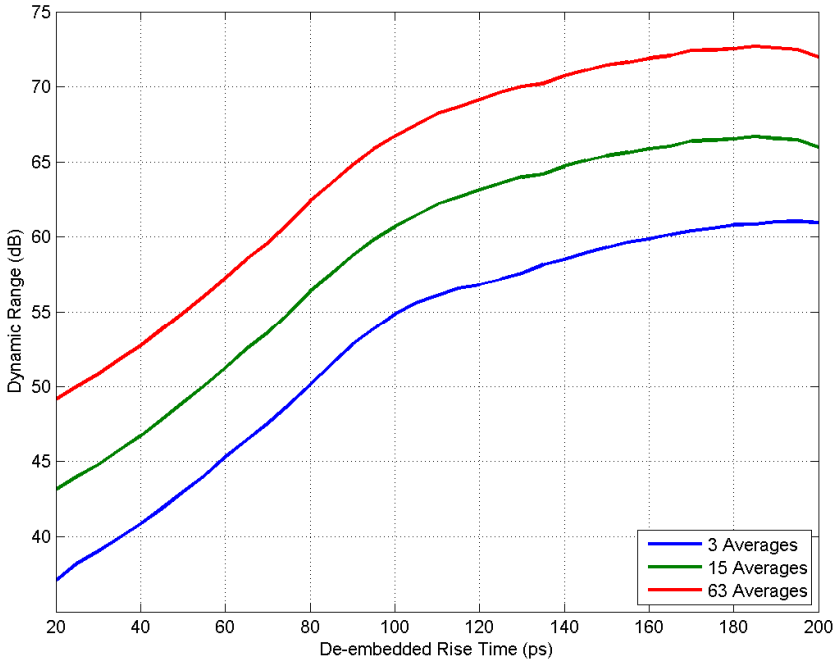

Figure 2.52. Dynamic Range With De-embedding

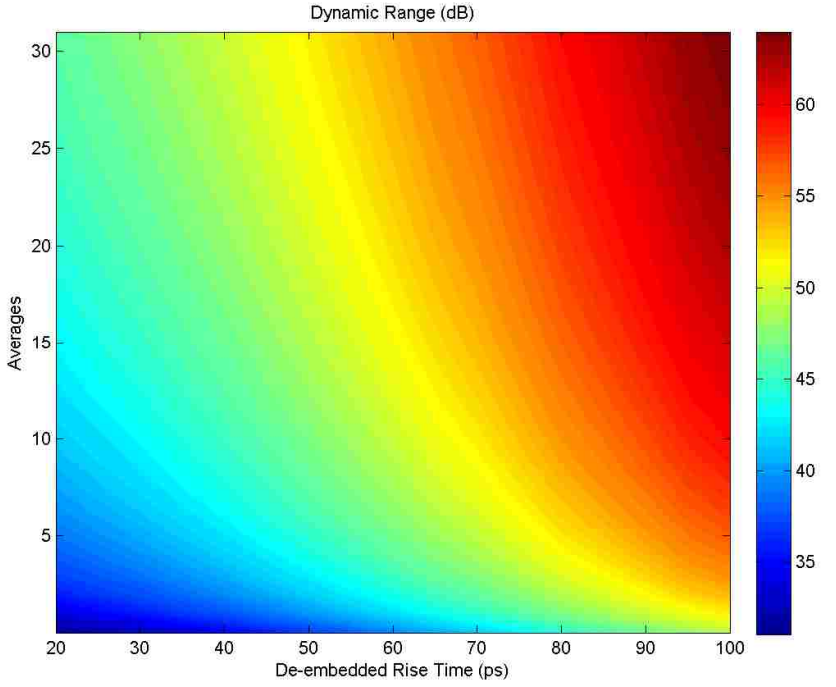

Figure 2.53. Dynamic Range With De-embedding

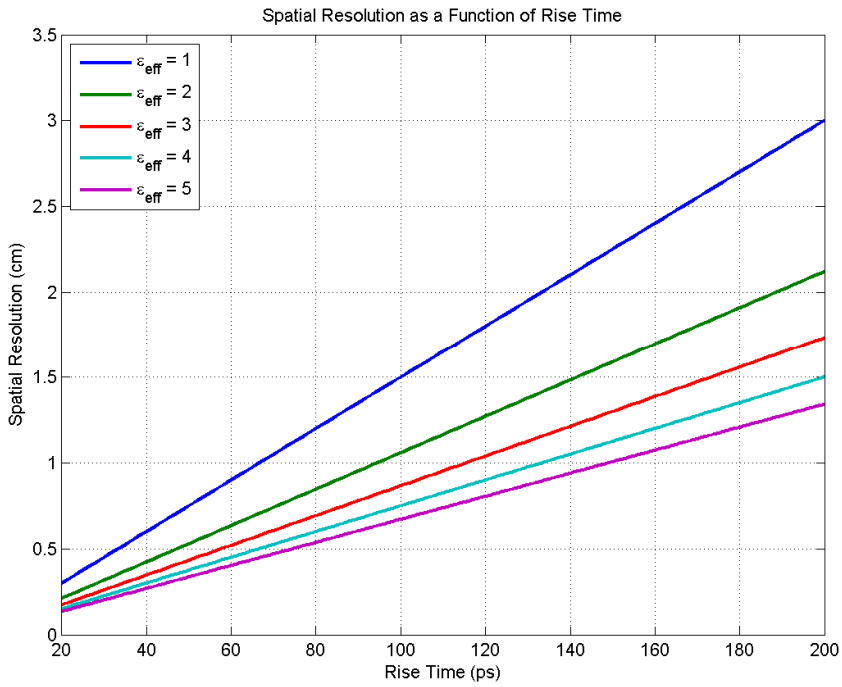

Figure 2.54. Spatial Resolution as a Function of Rise Time

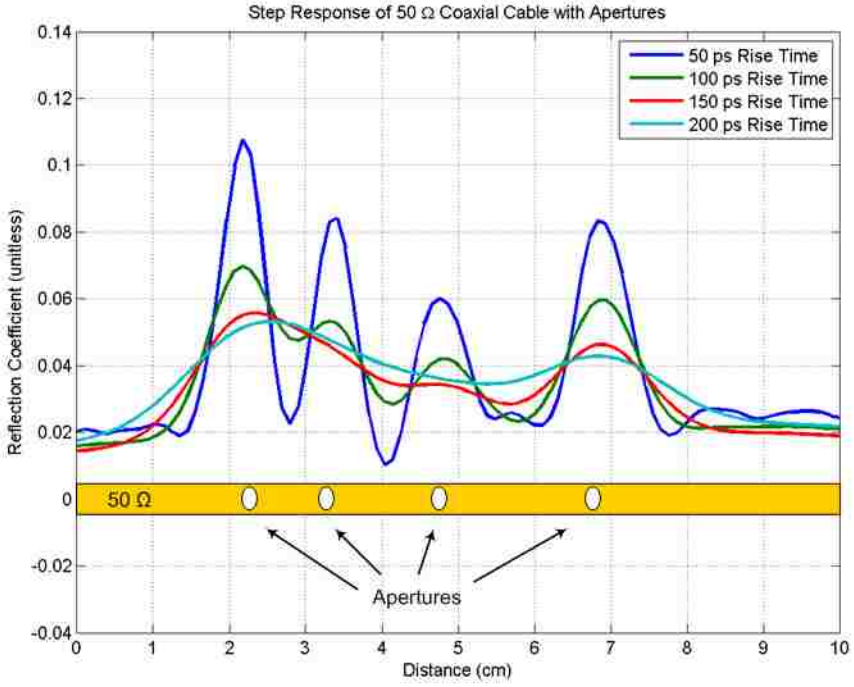

Figure 2.55. Step Response of a 50 Ω Coaxial Cable with Apertures

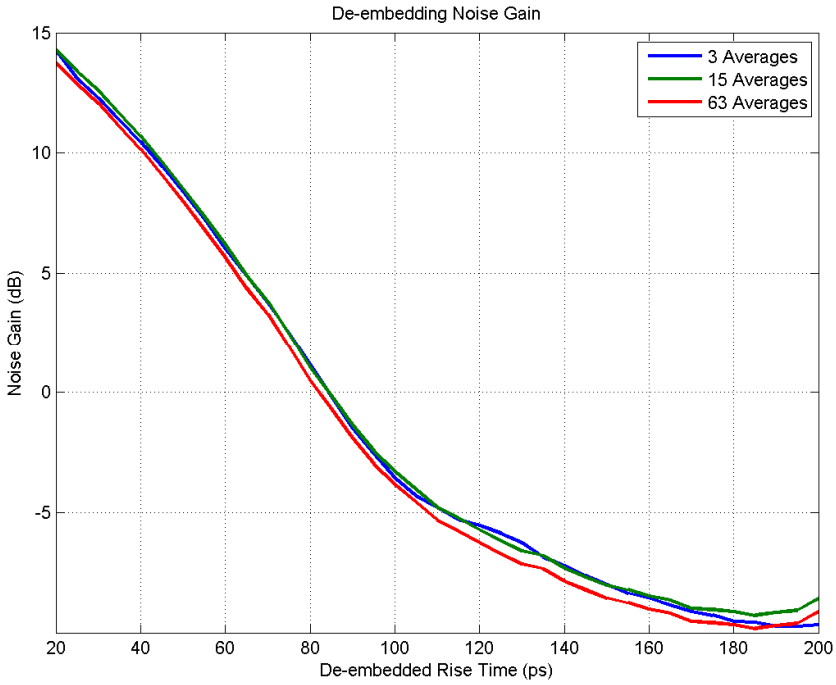

Figure 2.56. Noise Gain With De-embedding

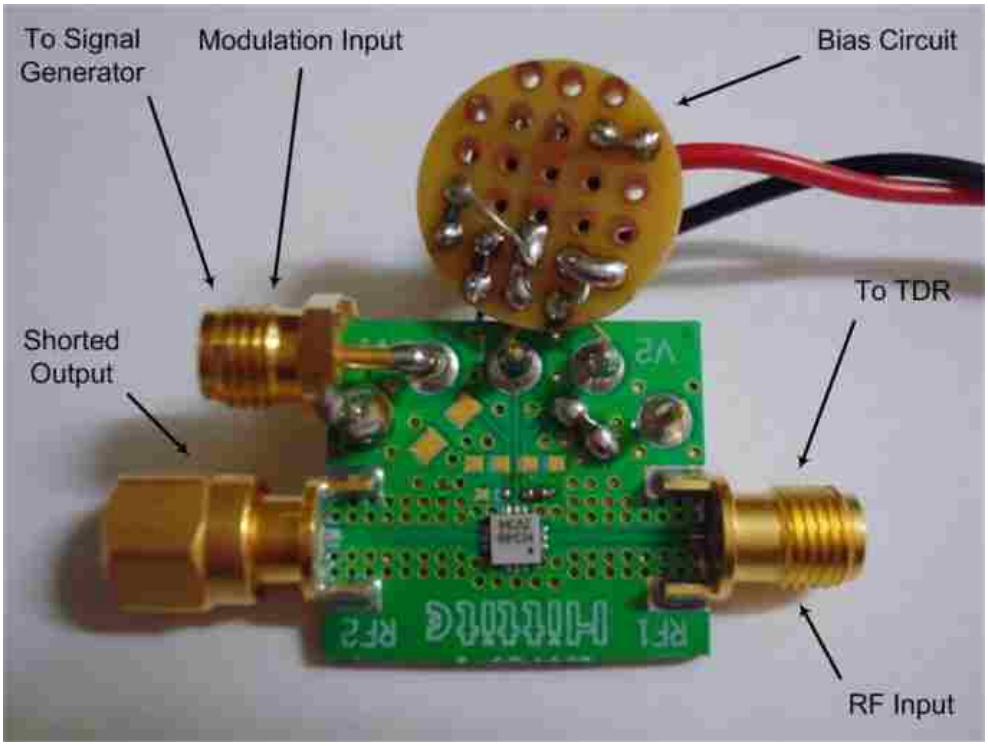

Figure 2.57. HMC346 Voltage Variable Attenuator

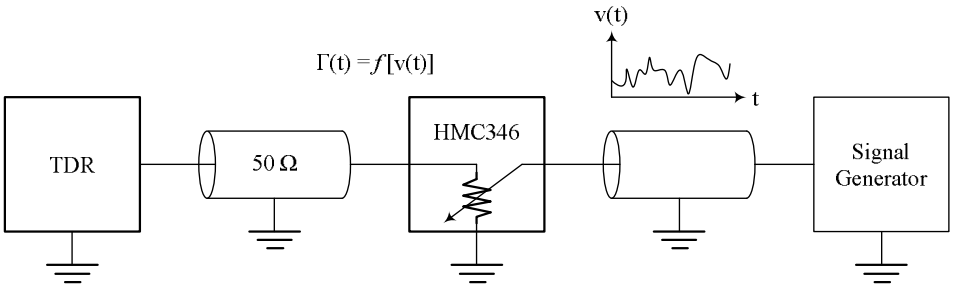

Figure 2.58. Setup for Evaluation of Acquisition Rate

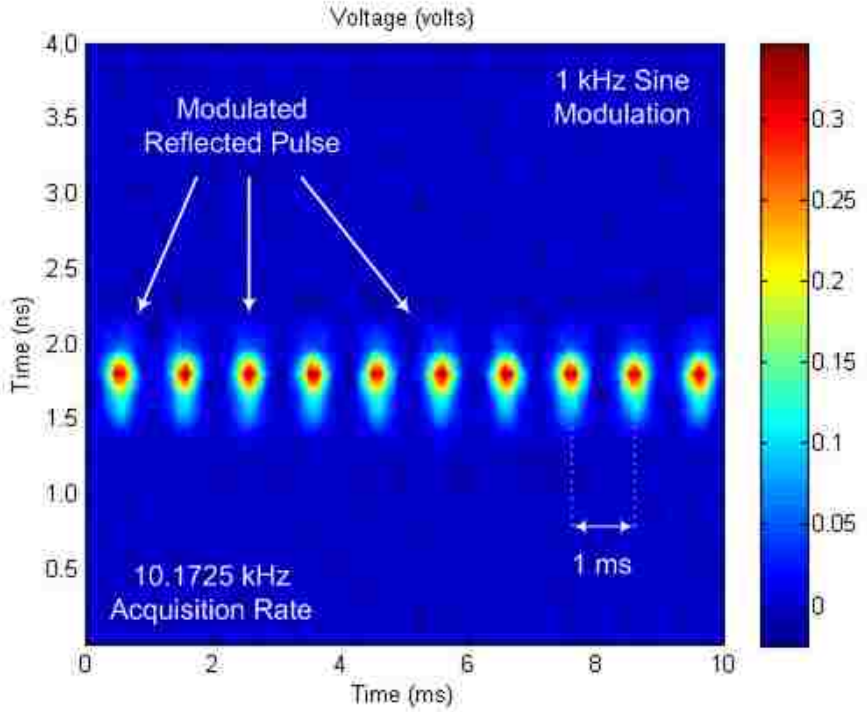

Figure 2.59. Impulse Response for 1 kHz Sine Modulation

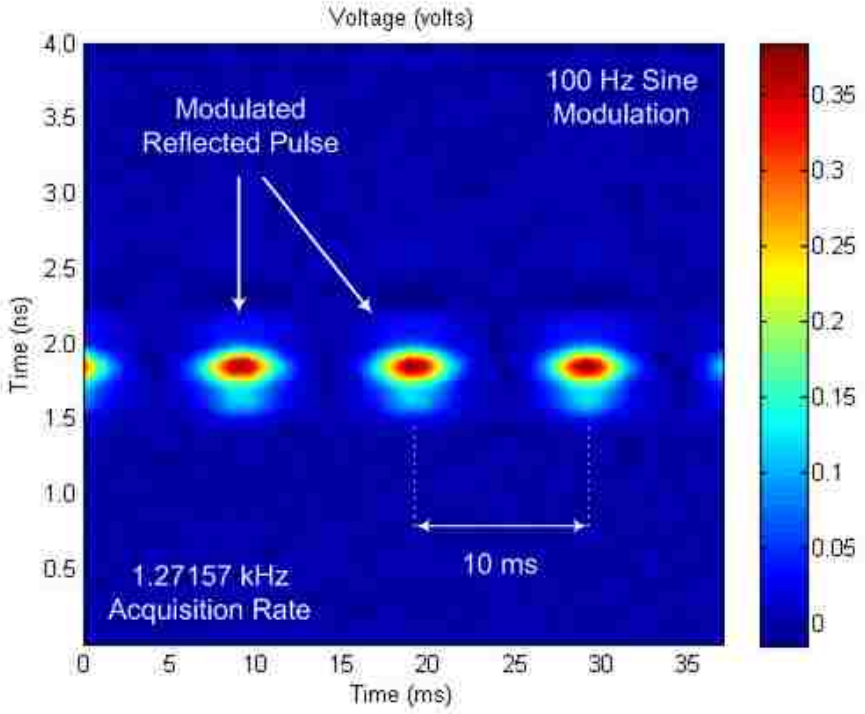

Figure 2.60. Impulse Response for 100 Hz Sine Modulation

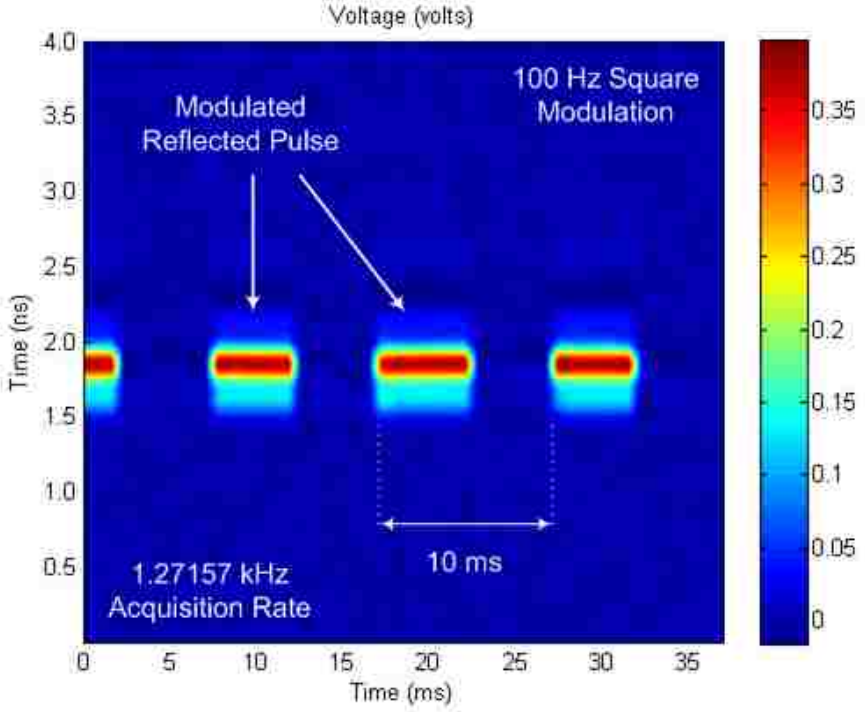

Figure 2.61. Impulse Response for 100 Hz Square Modulation

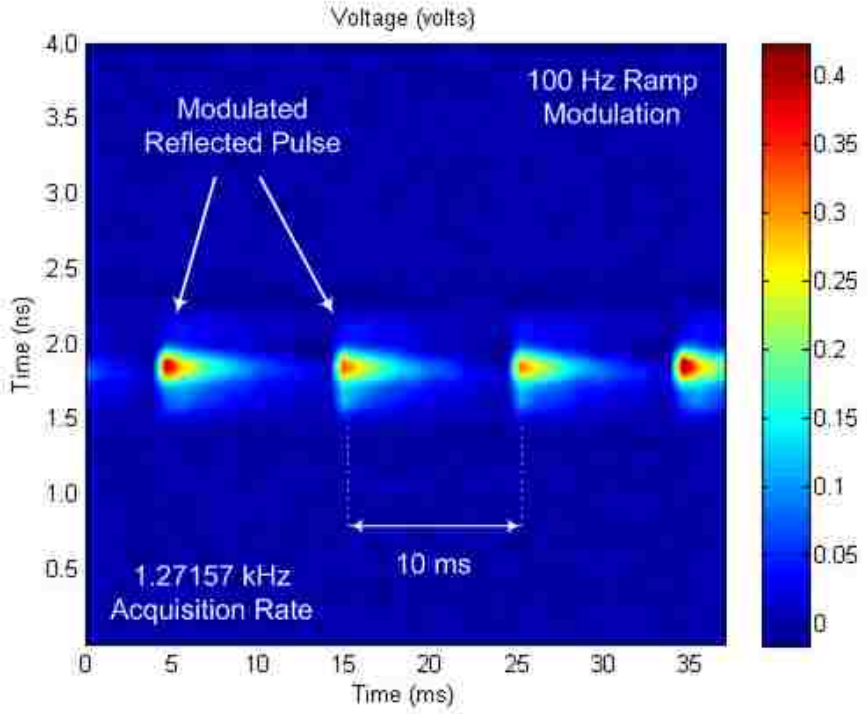

Figure 2.62. Impulse Response for 100 Hz Ramp Modulation

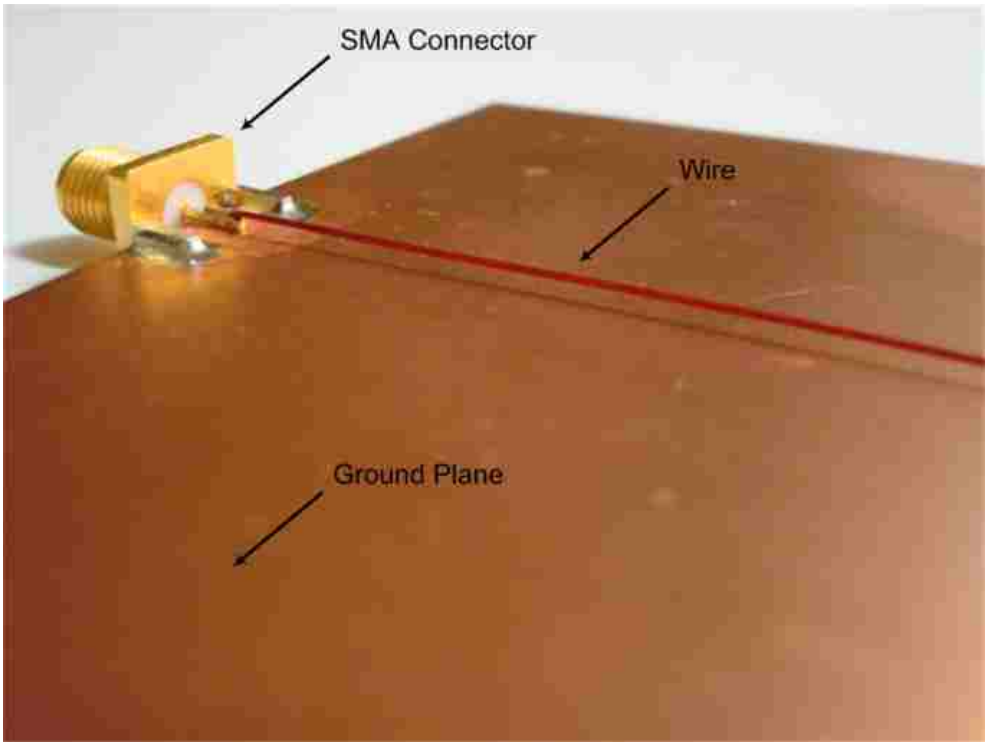

Figure 2.63. Wire Over Ground Plane

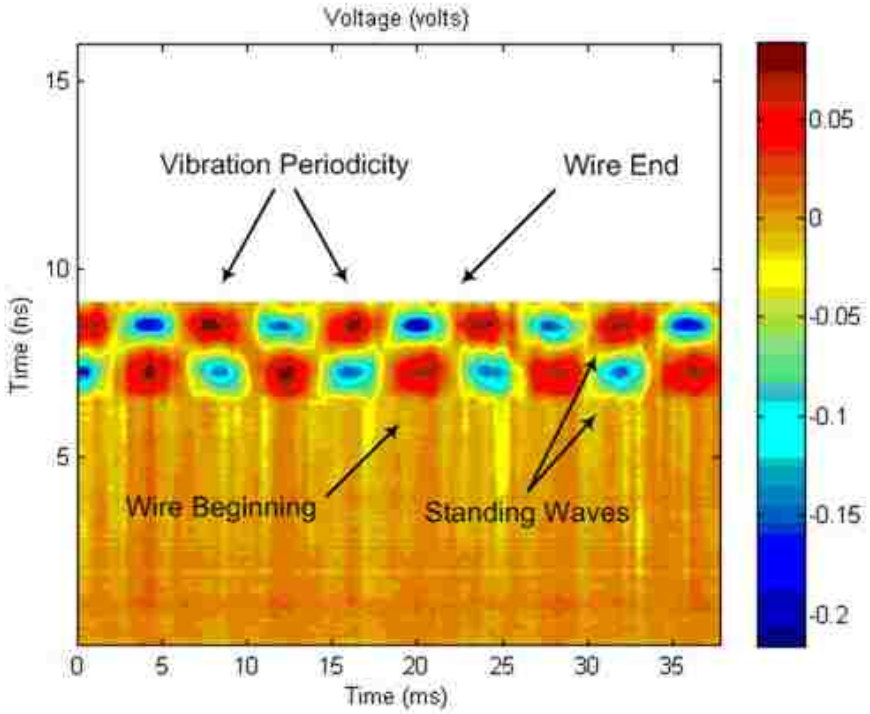

Figure 2.64. Step Response of Wire Over Ground Plane

3. APPLICATION

The TDR was applied in the monitor of a crack sensor embedded in a bridge column during a 60 second shake-table experiment at the Hyogo Earthquake Engineering Research Center in Japan. The bridge column is shown in Figure 3.1. The sensor was embedded in the southwest side of the bridge column as shown in Figure 3.2. The objective of the experiment was to resolve and monitor crack-induced changes in the voltage reflected from the sensor throughout the duration of the experiment. The sensor was monitored at an acquisition frequency of 385 Hz. Each acquisition was acquired in 1.6 milliseconds. The sensor was monitored from the beginning of the experiment, however the acceleration of the column was insignificant until approximately 13 seconds after the beginning of the experiment. The longitudinal acceleration and voltage reflected from the sensor are plotted in Figure 3.2 as a function of time. From Figure 3.2 it can be seen that the changes in the voltage reflected from the sensor correspond to the changes in the acceleration very well. The changes in the voltage reflected from the sensor were insignificant during the first 13 seconds and last 5 seconds of the experiment, which was consistent with the insignificant acceleration during those times. The changes in the voltage reflected from the sensor were significant during the time from 13 to 50 seconds into the experiment, which was consistent with the significant acceleration during that time. The changes in the voltage reflected from the sensor were greatest during the time from 13 to 25 seconds into the experiment, which was consistent with the strong acceleration during that time [7]. From Figure 3.2 it can also be seen that the only significant localized increase in the voltage reflected from the sensor occurred 0.35 meters from the beginning of the sensor, indicating that a single crack crossed the sensor at this location. In Figure 3.2 it appears that other cracks crossed the sensor, however careful examination of the photograph revealed that the cracks either crossed the portion of the sensor which was not embedded in the column or did not penetrate the grouting material because it was too strong. It should be noted that the TDR used in the experiment was an earlier model of the one discussed herein. Its design and operation were slightly different, but its concept was the same. Further information on the shake-table experiment can be found in [7].

Figure 3.1. Shake-Table Experiment

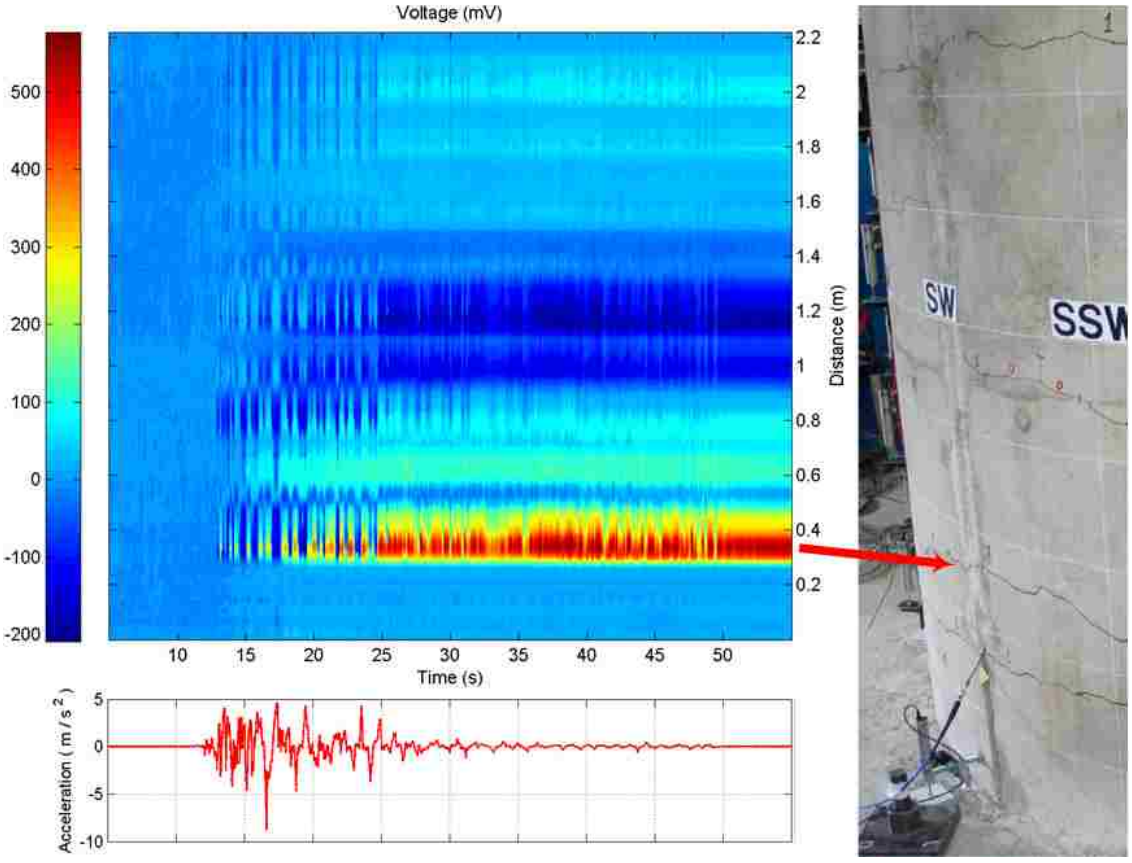

Figure 3.2. Results of Shake-Table Experiment

4. SPECIFICATIONS

The primary specifications of the TDR are listed in Table 4.1. The RMS noise and dynamic range are complex functions of the de-embedded rise time and so are specified for the 0 dB and 6 dB rise times. The 0 dB rise time is the de-embedded rise time for which the RMS voltage of the noise is equal to the RMS voltage of the noise without de-embedding. The 6 dB rise time is the de-embedded rise time for which the RMS voltage of the noise is twice the RMS voltage of the noise without de-embedding. The noise gain with de-embedding is discussed section 2.7.3 and plotted in Figure 2.56.

Table 4.1. Primary TDR Specifications

Amplitude of Impulse Reflected from Short Termination	- 974 mV
Dynamic Range for 60 ps De-embedded Rise Time	39 dB + 10 $\log_{10}$ (Averages + 1)
Dynamic Range for 85 ps De-embedded Rise Time	45 dB + 10 $\log_{10}$ (Averages + 1)
De-embedded Rise Time for 6 dB Noise Gain	60 ps
De-embedded Rise Time for 0 dB Noise Gain	85 ps
RMS Voltage of Noise for 85 ps De-embedded Rise Time	$0.0053\ \text{V} \div \sqrt{\text{Averages} + 1}$
RMS Reflection Coefficient of Noise for 85 ps De-embedded Rise Time	$0.0054 \div \sqrt{\text{Averages} + 1}$
Real-Time Sample Rate	125 MS/s
Equivalent-Time Sample Rate	12.5 GS/s, 25 GS/s, 50 GS/s, or 100 GS/s
Serial Baud Rate	8 MBaud
Memory Depth in Single Acquisition Mode	8 kS
Memory Depth in Multiple Acquisition Mode	128 MS
Maximum Signature Acquisition Frequency	10.1725 kHz × 12.5 GS/s ÷ Equivalent-Time Sample Rate
Minimum Signature Acquisition Time	24.576 µs × Equivalent-Time Sample Rate ÷ 12.5 GS/s

5. HARDWARE

A simplified block diagram of the TDR is shown in Figure 5.1. The blocks are discussed in the following subsections. The TDR acquisitions are controlled by the acquisition controller. The acquisition controller is controlled by a PC through a USB-RS232 converter. An acquisition is performed in the following sequence of events:

1. The acquisition controller triggers the impulse generator.
2. The impulse generator generates an impulse.
3. The generated impulse is amplified by the variable-gain amplifier.
4. The amplified impulse is transmitted to the DUT by a power splitter.
5. The transmitted impulse generates an impulse response from the DUT.
6. The impulse response is transmitted to the power limiter by the power splitter.
7. The power limiter amplifies the impulse response and limits its power to a safe level.
8. The amplified impulse response is sampled at 125 MS/s by the track-and-hold amplifier.
9. The analog samples of the impulse response are transmitted to the acquisition controller to be digitized and stored in memory.
10. The acquisition controller increments the impulse trigger delay by N, where N is 100 GS/s divided by the equivalent-time sample rate.
11. Events 1-10 are repeated 1024 times divided by N, where N is 100 GS/s divided by the equivalent-time sample rate.

The TDR enclosure is shown in Figure 5.2. The enclosure contains the following: one measurement port, seven expansion ports, one trigger port, one USB port, one CompactFlash slot, one power inlet, two intake fans, two outtake fans, two handles, and twenty shock absorbers. When opening the enclosure the lid should be removed with extreme care. Tilting the lid during removal can damage the internal electronics.

5.1. ACQUISITION CONTROLLER

The acquisition controller controls the acquisition by triggering the impulse generator, digitizing the analog samples from the track-and-hold amplifier, storing the digitized samples from the ADC, and communicating with the PC. A simplified block diagram of the acquisition controller is shown in Figure 5.3. The primary components of the acquisition controller are its FPGA, ADC, SDRAM, CompactFlash socket, delay line, and flash ROM. These primary components are discussed in the following subsections. A photograph of the acquisition controller is shown in Figure 5.4. The acquisition controller schematic is shown in Figure 5.5.

5.1.1. FPGA. The acquisition controller logic was implemented with a Xilinx Spartan-3 FPGA having 400 k gates. The logic was custom designed with the Verilog hardware description language and is discussed in section 6.3.

5.1.2. ADC. The analog samples from the track-and-hold amplifier are digitized at 125 MS/s with a 12-bit ADC from Linear Technology. The track-and-hold amplifier is discussed in section 5.6. The ADC is driven by a differential ADC driver from Linear Technology. The ADC driver is configured to have a voltage gain of two.

5.1.3. SDRAM. The digitized 12-bit samples from the ADC are temporarily stored in a 192 MB array of Micron SDRAM before being transmitted to a PC or stored on a CompactFlash card. The 192 MB SDRAM array is formed by paralleling three 64 MB SDRAM ICs. The data width of the SDRAM array is 24 bits which is equal to two 12-bit samples. All SDRAM functions, including initialization and refresh, are handled in the FPGA by the custom SDRAM controller module discussed in section 6.3.18.

5.1.4. Flash ROM. The acquisition parameters are stored in a 1 Mb Numonyx serial flash ROM. All flash ROM functions are handled in the FPGA by the custom flash ROM controller module. The ROM controller module is discussed in section 6.3.24. The acquisition parameters are read from the flash ROM at power-on and stored in a register bank in the FPGA from which they are read by a PC and other modules in the FPGA. The acquisition parameters stored in the flash ROM may be rewritten over a PC.

5.1.5. Delay Line. The sequential, equivalent-time sampling is accomplished by sequentially delaying the impulse trigger, and thus the generated impulse, with a programmable delay line from Micrel. Generation of the impulse is discussed in section 5.2. The impulse trigger may be delayed in 10, 20, 40, or 80 ps increments to achieve a respective equivalent-time sample rate of 12.5, 25, 50, or 100 GS/s.

5.1.6. CompactFlash. After being temporarily stored in SDRAM, the 12-bit samples from the ADC are written to a CompactFlash card if the acquisition controller is in multiple acquisition mode. The CompactFlash socket supports CompactFlash cards of types I and II. The CompactFlash card is configured to operate in True IDE mode and transfer data in its highest supported PIO mode. All CompactFlash functions are handled in the FPGA by the custom CompactFlash controller module discussed in section 6.3.12.

5.2. IMPULSE GENERATOR

The impulse generator generates an impulse when triggered by the acquisition controller. The generated impulse is shown in Figure 5.7. The impulse is generated by coupling a step pulse with a 1 pF capacitor. The step pulse is generated by an 11.3 Gb/s limiting amplifier having a measured rise time of 20 ps. The limiting amplifier is preceded by a 4.25 Gb/s CML buffer in order to minimize the rise time of the impulse trigger and maximize the bandwidth of the generated impulse. A photograph of the impulse generator is shown in Figure 5.6. The impulse generator schematic is shown in Figure 5.8.

5.3. VARIABLE-GAIN AMPLIFIER

The variable-gain amplifier varies the amplitude of the impulse generated by the impulse generator. The amplitude of the impulse is varied by an HMC424LP3 broadband digital attenuator which can be set manually via a six position DIP switch or digitally via an eight pin header. The attenuated impulse is amplified by a chain of three NBB-310 amplifiers. The NBB-310 was chosen for its high performance, low cost, and cascadability. RC networks were included between amplifiers to flatten the response by compensating for frequency-dependent loss. The frequency response at maximum gain

is shown in Figure 5.10. From Figure 5.10 it can be seen that the bandwidth of the variable-gain amplifier is approximately 12 GHz. The variable-gain amplifier contains additional logic circuitry to translate the 3.3 volt logic inputs to the -5 volt logic of the digital attenuator. Although the gain of the amplifier is variable, it is set to its maximum value by the DIP switch. A photograph of the variable-gain amplifier is shown in Figure 5.9. The variable-gain amplifier schematic is shown in Figure 5.11.

5.4. POWER LIMITER

The power limiter limits the power input to the track-and-hold amplifier. The power is limited by an NBB-310 amplifier followed by an HMC424LP3 broadband digital attenuator. A digital attenuator was used so that the power limit would be variable. The power limit is set manually via a six position DIP switch. The NBB-310 was chosen for its low cost and high performance. An RC network is included after the NBB-310 to flatten the response by compensating for frequency-dependent loss. The frequency response at maximum output power is shown in Figure 5.13. From Figure 5.13 it can be seen that the bandwidth of the power limiter is approximately 13 GHz. Although the power of the power limiter is variable, it is set to its maximum value by the DIP switch. A photograph of the power limiter is shown in Figure 5.12. The power limiter schematic is shown in Figure 5.14.

5.5. POWER SPLITTER

A ZFRSC-123+ power splitter from Mini-Circuits is used to provide a broadband, matched path connecting the DUT to the transmitting and receiving ports of the TDR. A photograph of the power splitter is shown in Figure 5.15.

5.6. TRACK-AND-HOLD AMPLIFIER

A 1321TH track-and-hold amplifier from Inphi is used to acquire analog samples of the impulse response for the acquisition controller ADC to digitize. When clocked, the track-and-hold amplifier samples its input voltage and outputs the sampled voltage until it is re-clocked. The operation of the track-and-hold amplifier is illustrated in Figure 5.17. The track-and-hold amplifier supports an additional mode of operation

called Full Track Mode in which it ignores its clock inputs and operates as a unity gain amplifier. The track-and-hold amplifier was purchased on an evaluation board due to its high cost and high vulnerability to electrostatic discharge (ESD). A photograph of the track-and-hold amplifier evaluation board is shown in Figure 5.16. The frequency response of the track-and-hold amplifier evaluation board is shown in Figure 5.18. The frequency response was measured with the track-and-hold amplifier in the Full Track Mode of operation.

5.7. USB-RS232 CONVERTER

An FT2232H USB-RS232 converter from FTDI is used to convert USB commands from a PC to RS232 commands interpreted by the acquisition controller FPGA. A photograph of the USB-RS232 converter is shown in Figure 5.19.

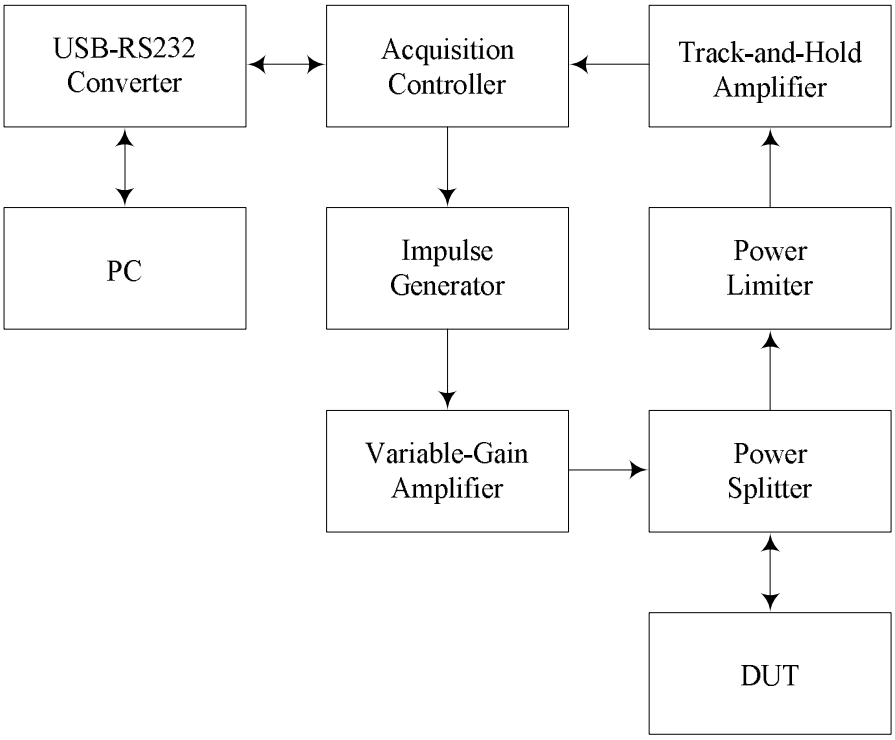

Figure 5.1. Simplified TDR Block Diagram

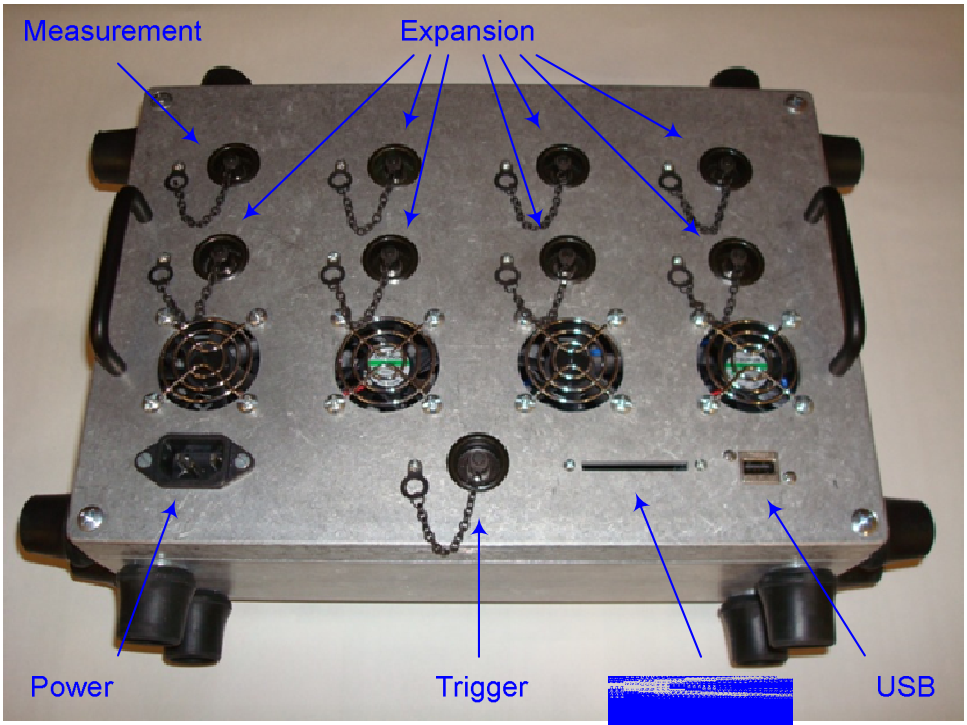

Figure 5.2. TDR Enclosure

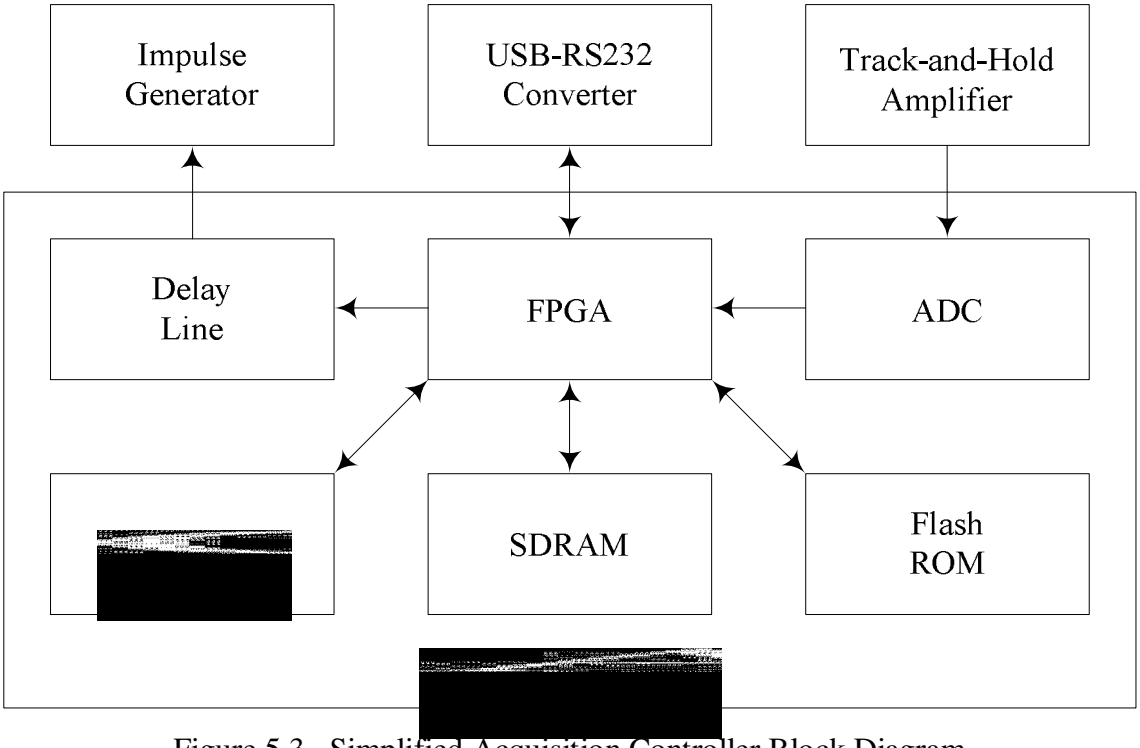

Figure 5.3. Simplified Acquisition Controller Block Diagram

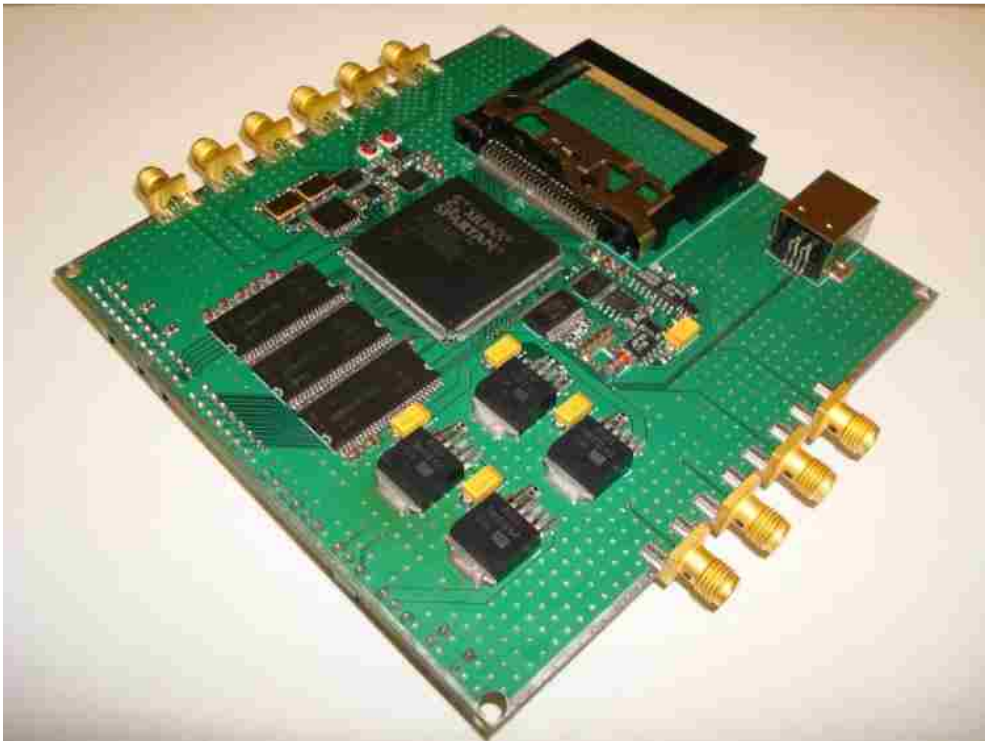

Figure 5.4. Acquisition Controller Printed Circuit Board

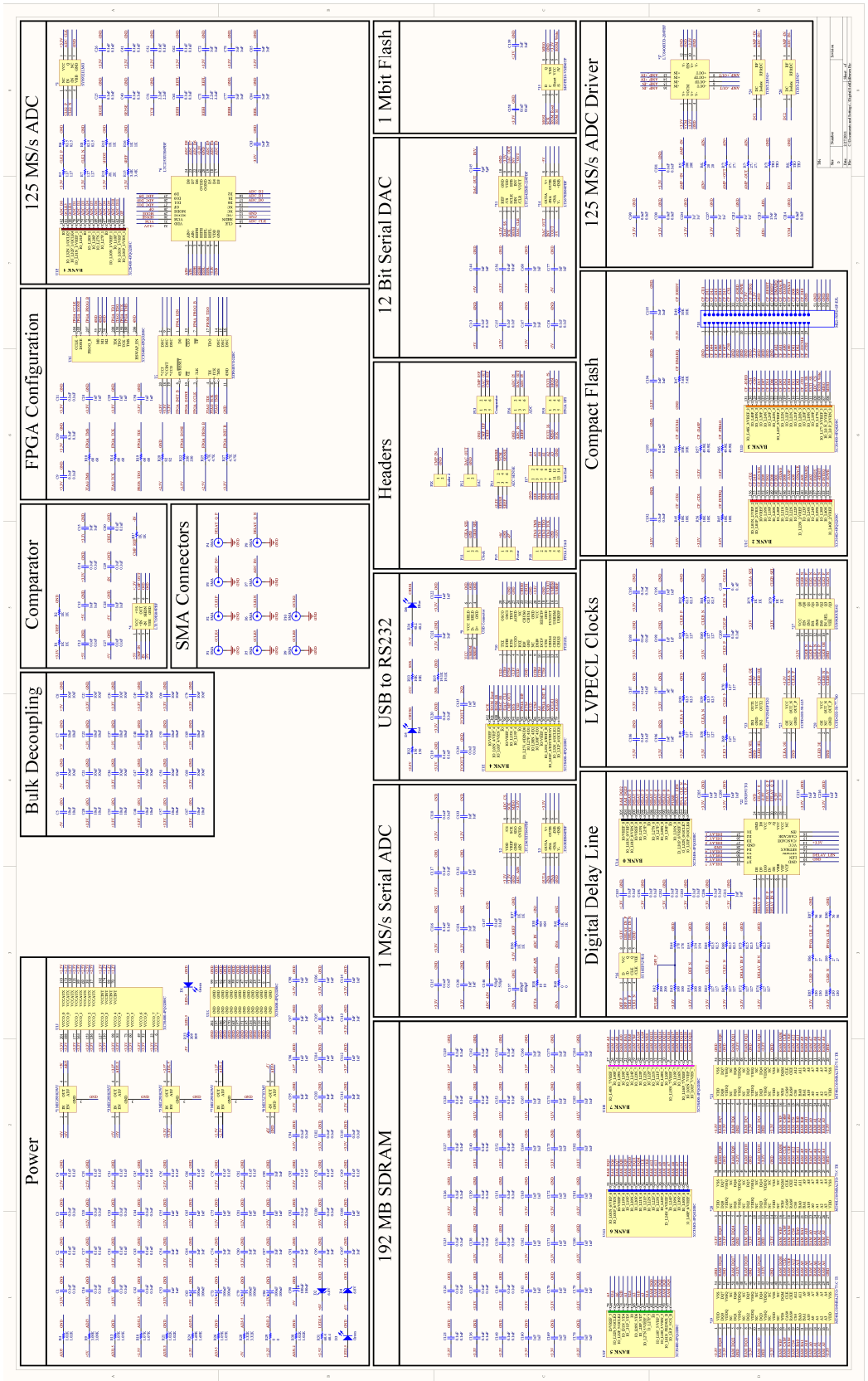

Figure 5.5. Acquisition Controller Schematic

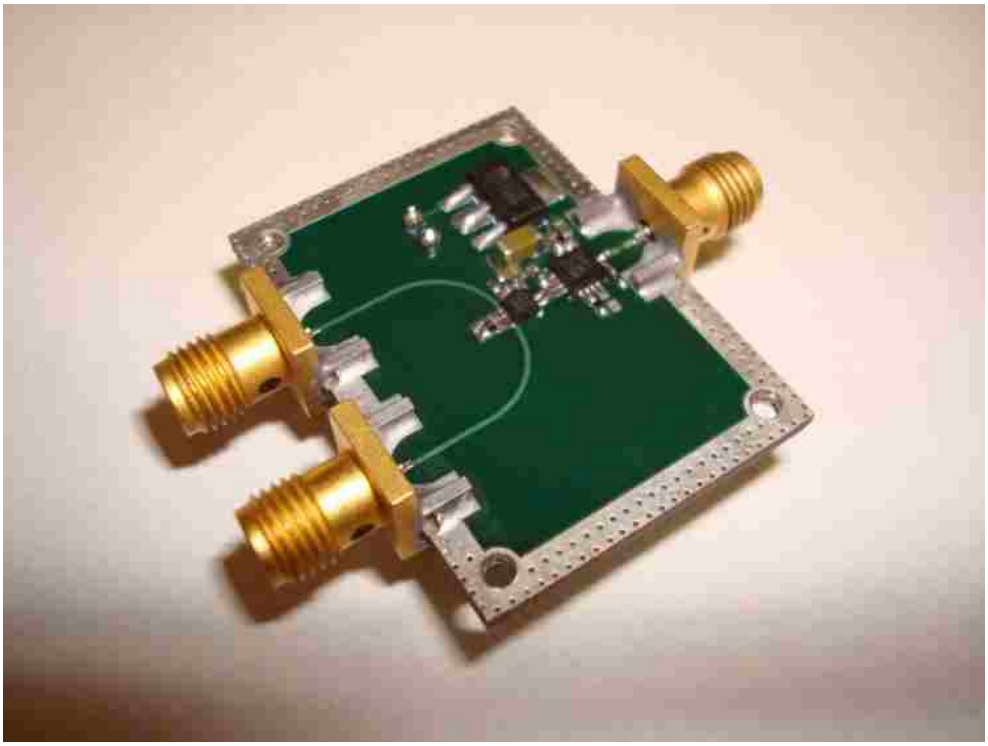

Figure 5.6. Impulse Generator Printed Circuit Board

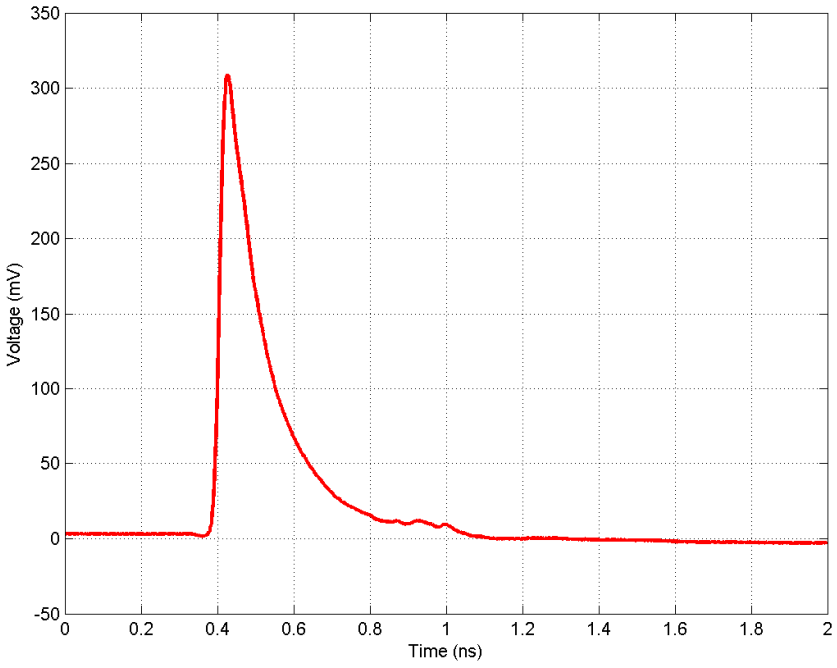

Figure 5.7. Generated Impulse

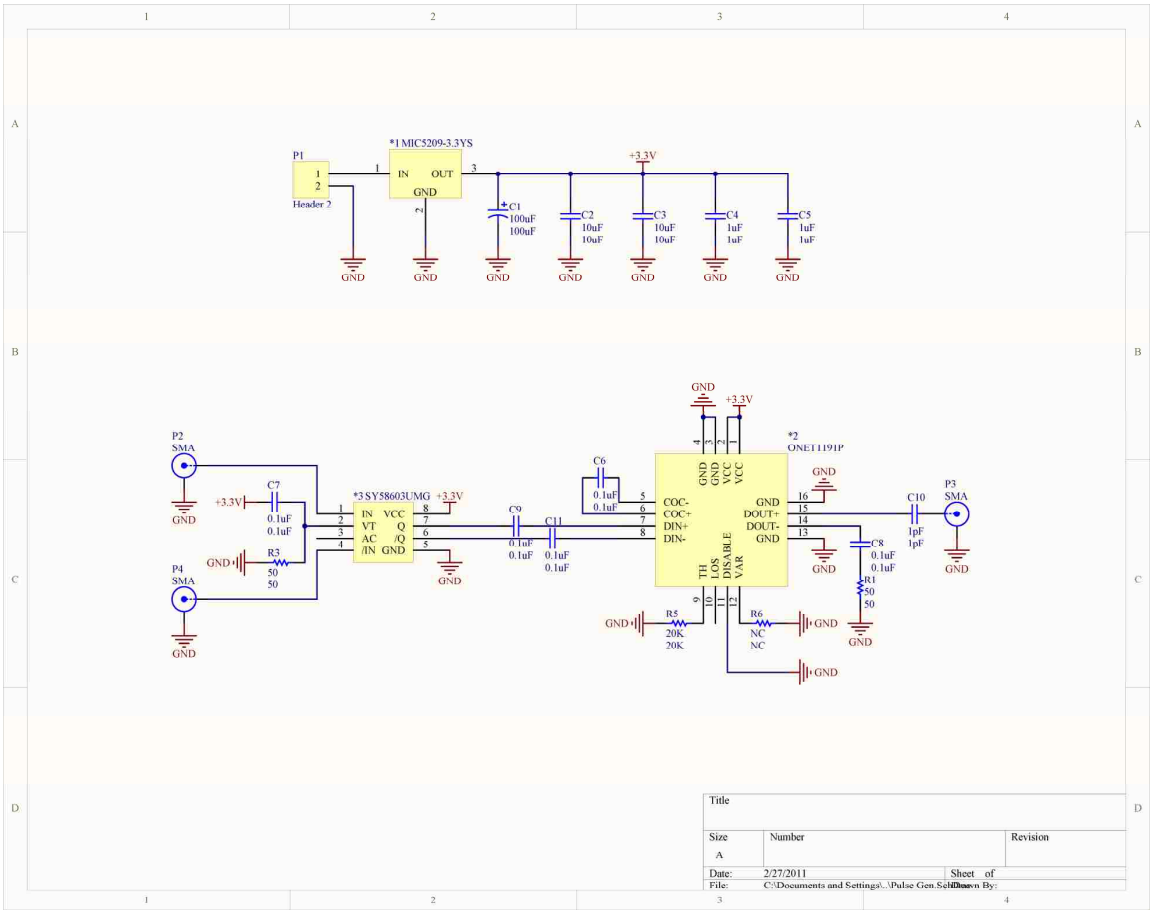

Figure 5.8. Impulse Generator Schematic

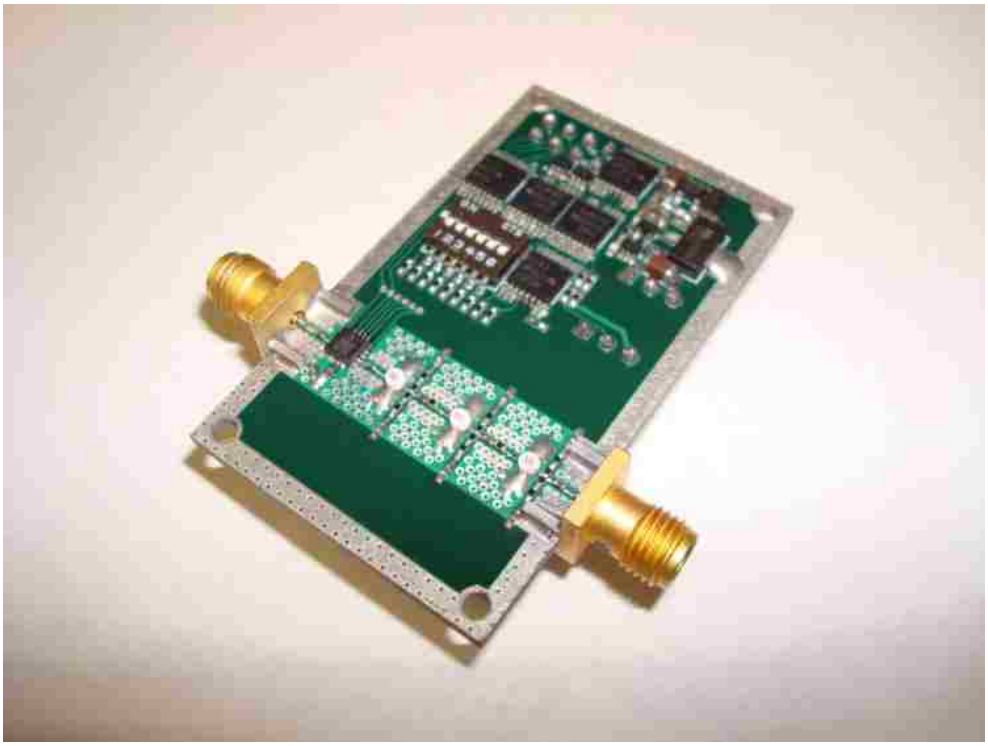

Figure 5.9. Variable-Gain Amplifier Printed Circuit Board

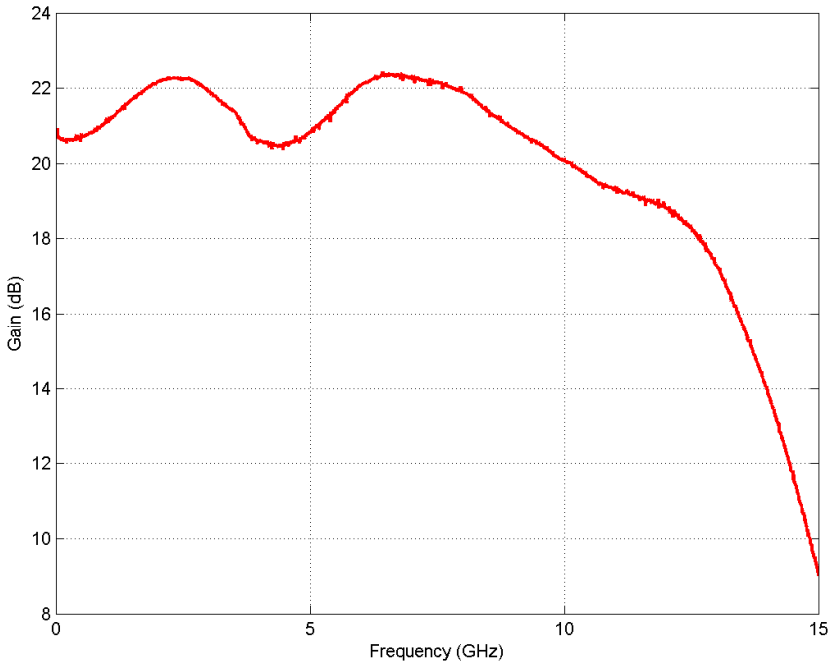

Figure 5.10. Frequency Response at Maximum Gain

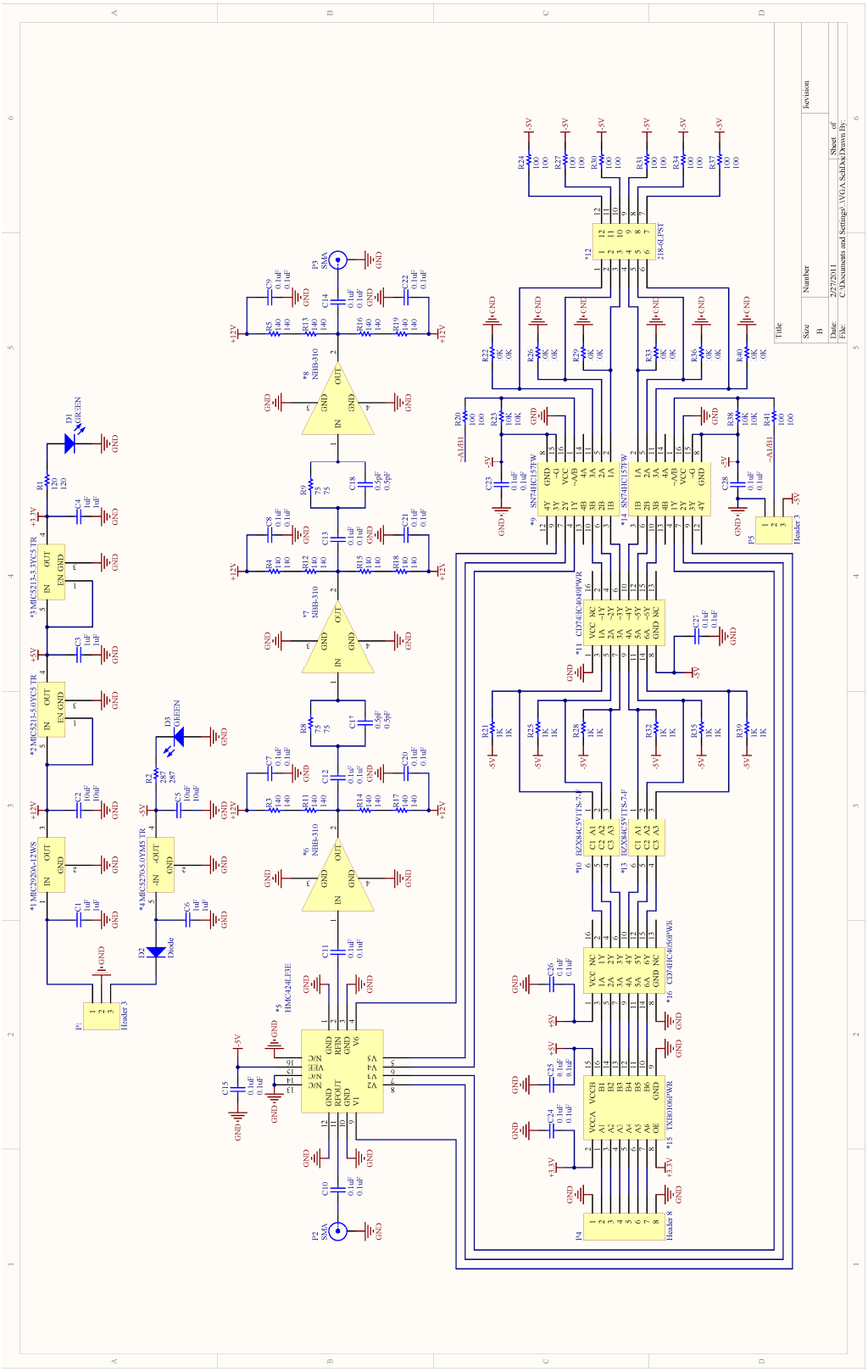

Figure 5.11. Variable-Gain Amplifier Schematic

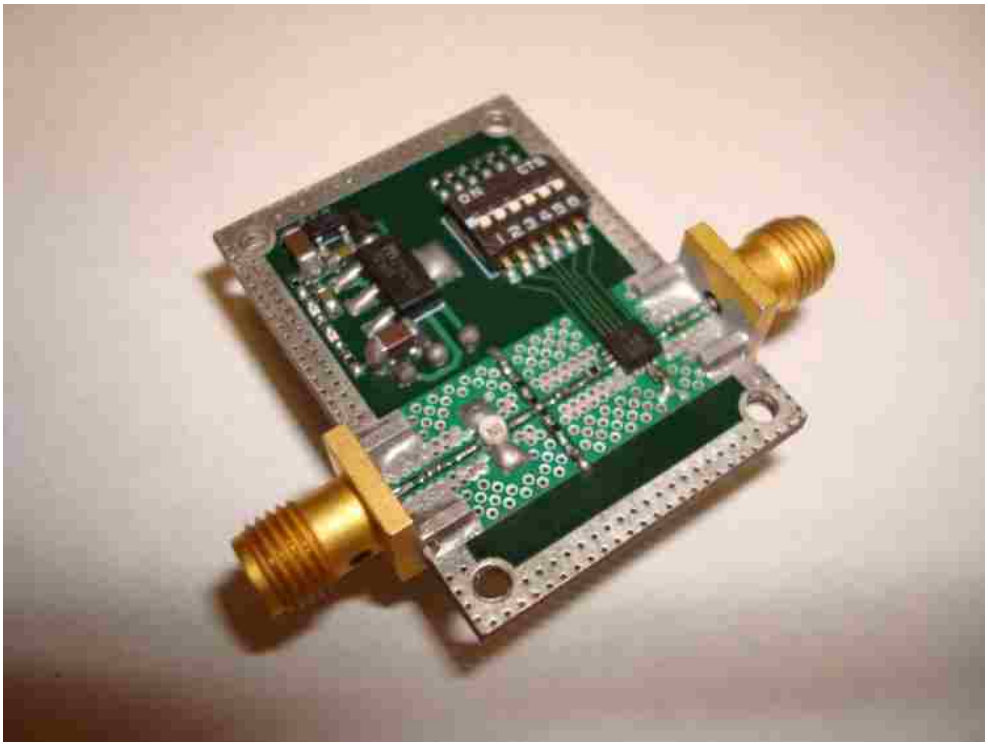

Figure 5.12. Power Limiter Printed Circuit Board

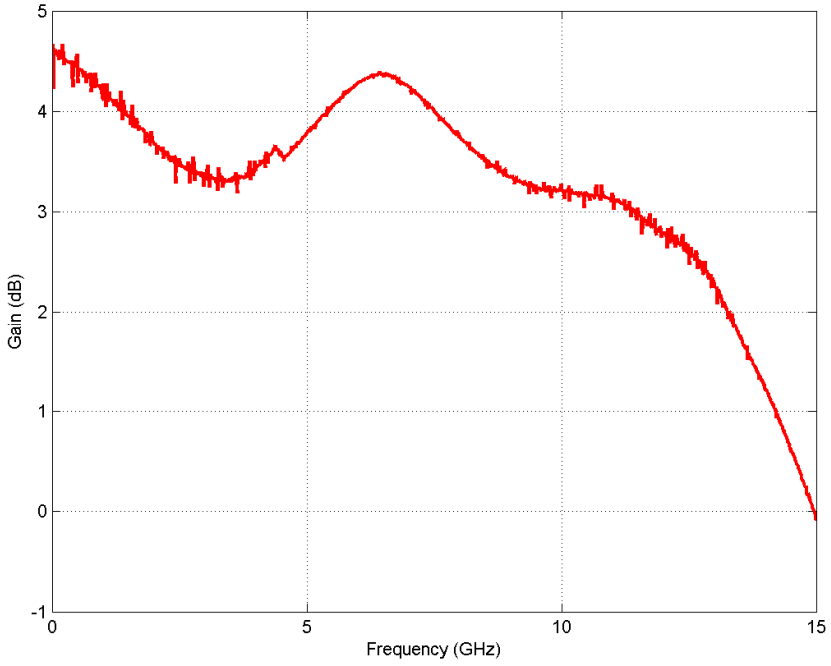

Figure 5.13. Frequency Response at Maximum Output Power

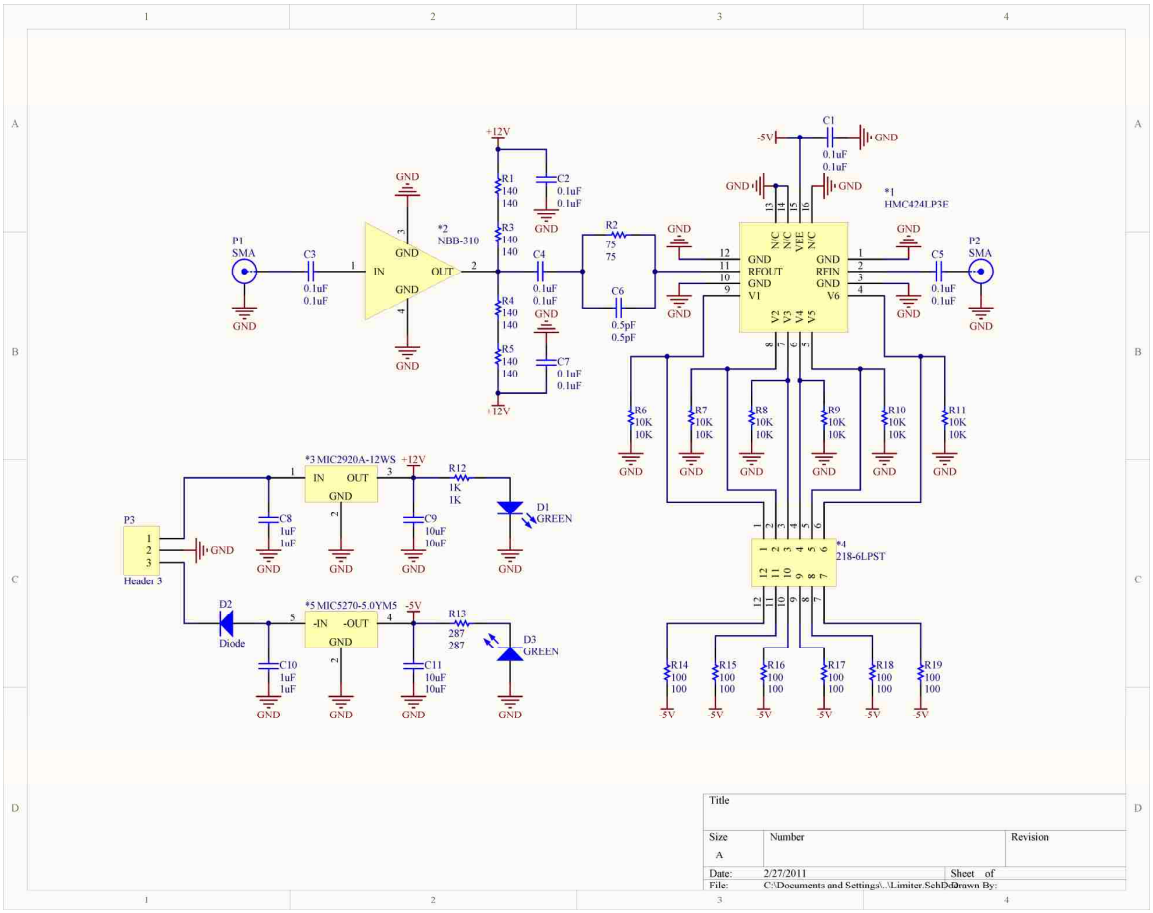

Figure 5.14. Power Limiter Schematic

Figure 5.15. Power Splitter

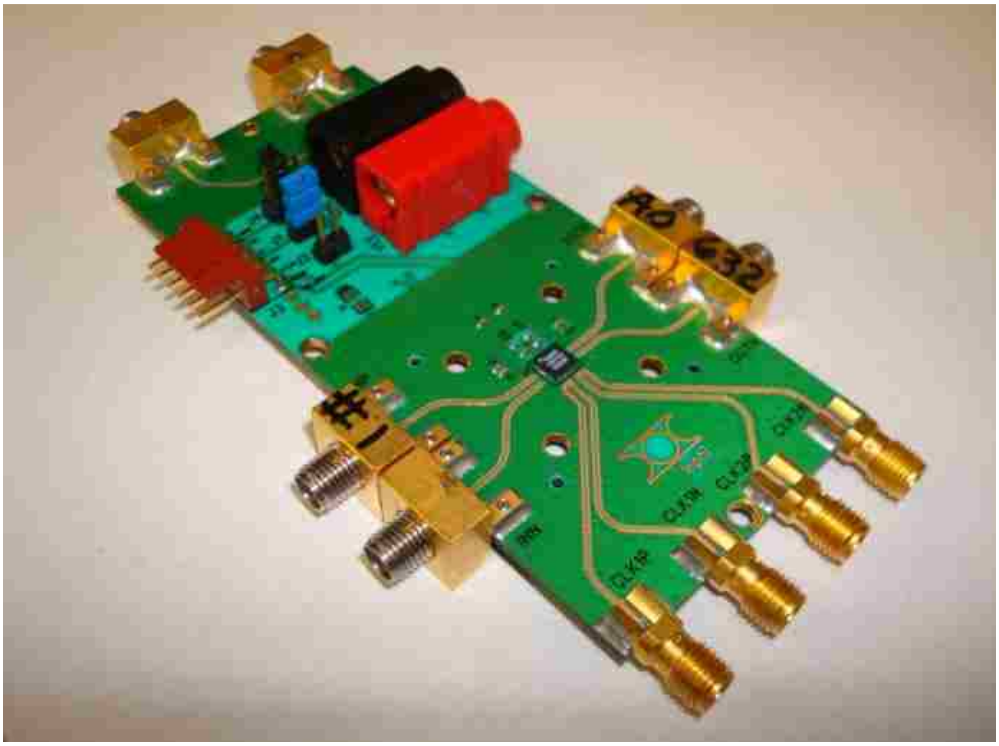

Figure 5.16. Track-and-Hold Amplifier Evaluation Board

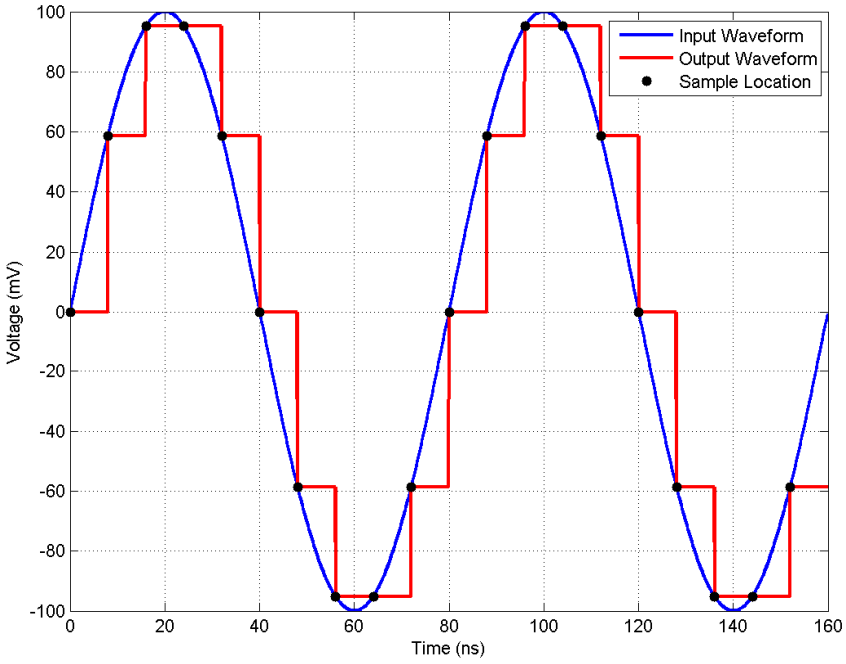

Figure 5.17. MATLAB Illustration of Track-and-Hold Operation

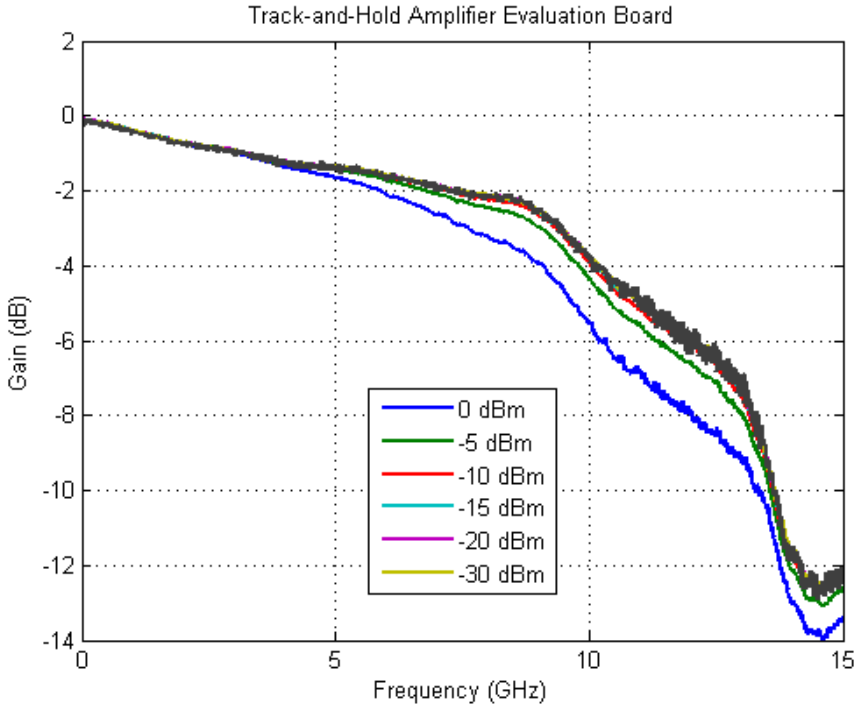

Figure 5.18. Frequency Response of Track-and-Hold Amplifier Evaluation Board

Figure 5.19. USB-RS232 Converter

6. SOFTWARE

6.1. HEXEDIT

In the multiple acquisition mode the acquired data is stored in binary on an unformatted CompactFlash card. The data stored on the CompactFlash card can be viewed and transferred to the PC using a CompactFlash card reader and a free hex editor called HEXEdit. A screenshot of the HEXEdit hex editor is shown in Figure 6.1. The data can be viewed with HEXEdit by reading the appropriate number of sectors at the appropriate starting sector of the disk associated with the CompactFlash card. The viewed data can be transferred to the PC by saving it as a binary file with a .bin extension. The saved data can be processed in LabVIEW by selecting the binary file with the Data File Path control on the Process tab of the user interface. The Data File Path control and Process tab are discussed in section 6.2.2.

6.2. LABVIEW

The TDR is monitored and controlled by a PC through a graphical user interface designed in LabVIEW. The graphical user interface consists of a control tab and a process tab. The control tab is the primary tab containing controls and indicators used in both the single acquisition mode and the multiple acquisition mode. A screenshot of the control tab is shown in Figure 6.2. The process tab is a secondary tab containing controls and indicators used only in the multiple acquisition mode. A screenshot of the process tab is shown in Figure 6.3. The control and process tabs of the graphical user interface are discussed in the following subsections.

6.2.1. Control Tab. The controls of the control tab are listed in Table 6.1. The Port control is used to select the port ID of the virtual COM port associated with the FTDI chip of the USB-RS232 converter. The Baud Rate control is used to specify the baud rate of the virtual COM port associated with the FTDI chip of the USB-RS232 converter. The Storage Mode control is used to select the acquisition mode. The Sample Rate (GS/s) control is used to select the equivalent-time sample rate. The Pulse Period (ns) control is used to specify the period of the impulse stimulus. The Delay (ns) control is used to specify the time delay of the acquisition. The Record Length (ns) control is

used to specify the time length of the acquisition. The Acquisitions Off control is used to specify the burst separation of the multiple acquisition mode. The Acquisitions On control is used to specify the burst length of the multiple acquisition mode. The Starting Sector control is used to specify the sector at which the data written to the CompactFlash card will begin. The Sectors control is used to specify the amount of data written to the CompactFlash card. The Countdown (s) control is used to specify the number of seconds to delay the initiation of a measurement. The Smoothing Samples control is used to specify the number of samples in the moving average of the smoothing operation. The Smoothing Iterations control is used to specify the number of times to perform the smoothing operation. The Averages control is used to specify the number of averages to perform. The Rise Time (ps) control is used to specify the de-embedded rise time. The Units control is used to select the units of the processed acquisition. The Response Type control is used to select between the impulse and step responses. The Save File Path control is used to specify the path of the file to which the acquisition data will be written if the Save control is enabled. The Save control is used to enable writing of the acquisition data to the file specified by the Save File Path. The Normalize control is used to enable normalization. The De-embed control is used to enable de-embedding. The Load ROM control is used to enable loading of acquisition parameters to the flash ROM. The Smooth control is used to enable the smoothing operation. The Acquire Single control is used to initiate a single measurement. The Acquire Continuous control is used to continuously initiate measurements. The Armed control is used to arm the external measurement trigger. The Calibrate control is used to open the calibration wizard. The window control is used to enable the windowing operation. The Temperature Control control is used to enable the temperature controller of the delay module. The Cursor 1 control is used to enable Cursor 1. The Cursor 2 control is used to enable Cursor 2. The indicators of the control tab are listed in Table 6.2. The Temperature Good indicator is used to indicate when the temperature of the programmable delay line is stabilized. The Loading Registers indicator is used to indicate when the acquisition parameters are being loaded in the TDR registers. The Acquiring Data indicator is used to indicate when data is being acquired. The Processing Data indicator is used to indicate when data is being processed. The Saving Data

indicator is used to indicate when data is begin saved. The Loading ROM indicator is used to indicate when the acquisition parameters are being loaded in the flash ROM. The Progress indicator is used to indicate the progress of the acquisition. The Acquisition Number indicator is used to indicate the acquisition number for which data is being acquired. The Elapsed Time (s) indicator is used to indicate the amount of time which has elapsed during the measurement. The Sample Period (s) indicator is used to indicate the time separation of the acquired samples. The Samples Per Acquisition indicator is used to indicate the number of samples per acquisition. The Pulses Per Acquisition indicator is used to indicate the number of pulses over which an acquisition is performed. The Acquisition Time (s) indicator is used to indicate the period of time over which an acquisition is performed. The Acquisition Period (s) indicator is used to indicate the acquisition period during the burst. The Acquisition Frequency (Hz) indicator is used to indicate the acquisition rate during the acquisition burst. The Acquired Signatures indicator is used to indicate the number of signatures acquired during the measurement. The Averaged Acquired Signatures indicator is used to indicate the number of signatures acquired during the measurement after averaging. The Acquired Data (B) indicator is used to indicate the amount of data acquired during the measurement. The Acquisition Off Time (s) indicator is used to indicate the time separation of the acquisition bursts. The Acquisition On Time (s) indicator is used to indicate the time lengths of the acquisition bursts. The Averaged Acquisition Period (s) indicator is used to indicate the acquisition period after averaging. The Averaged Acquisition Frequency (Hz) indicator is used to indicate the acquisition rate after averaging. The Measurement Time (s) indicator is used to indicate the period of time over which the measurement is performed. The X1 indicator is used to indicate the X value of Cursor 1. The X2 value is used to indicate the X value of Cursor 2. The ΔX indicator is used to indicate the difference between the X values of Cursors 1 and 2. The Y1 indicator is used to indicate the Y value of Cursor 1. The Y2 indicator is used to indicate the Y value of Cursor 2. The ΔY indicator is used to indicate the difference between the Y values of Cursors 1 and 2. The Excess Reactance indicator is used to indicate the excess inductance or capacitance between the X values of Cursors 1 and 2. The XY Graph indicator is used to display the processed acquisition.

6.2.2. Process Tab. The controls of the process tab are listed in Table 6.3. The Baseline File Path control is used to specify the path of the file containing the baseline acquisition which is subtracted from the processed acquisitions when differencing is enabled. The Load File Path control is used to specify the path of the load calibration file which is used for normalization. The Short File Path control is used to specify the path of the short calibration file which is used for de-embedding. The Data File Path control is used to specify the path of the binary file containing the data acquired in the multiple acquisition mode. The Save Path control is used to specify the path to which the plots of the processed acquisitions will be saved. The Line Width control is used to specify the width of the line in the linear plots of the processed acquisitions. The Line Color control is used to specify the color of the line in the linear plots of the processed acquisitions. The Contour Levels control is used to specify the number of contour levels in the contour plots of the processed acquisitions. The Format control is used to specify the format in which the linear and contour plots of the processed acquisitions will be saved. The Process control is used to initiate the acquisition processing. The Difference control is used to enable acquisition differencing so that the baseline acquisition is subtracted from the processed acquisitions. The Linear Plots control is used to enable linear plotting of the processed acquisitions. The Contour Plots control is used to enable contour plotting of the processed acquisitions. The Grid On control is used to enable the grid in the linear plots of the processed acquisitions. The indicators of the process tab are listed in Table 6.4. The Progress indicator is used to indicate the progress of the linear plotting of the processed acquisitions. The Plot Number indicator is used to indicate the numbers of the plots in the linear plotting of the processed acquisitions. The Elapsed Time (s) indicator is used to indicate the amount of time which has elapsed during the acquisition processing.

6.3. VERILOG

The acquisition controller logic was designed with the Verilog hardware description language (HDL). All of the modules were custom designed and are described in the following subsections. The HDL code for each module is included in the appendix.

6.3.1. acq. The acq module controls the acquisition of samples from the adc module. The acq module is controlled by the acq_ctrl module. The acq module inputs are listed in Table 6.5. The acq module outputs are listed in Table 6.6. The code for the acq module is included in the appendix.

6.3.2. acq_ctrl. The acq_ctrl module controls the acq module. The acq_ctrl module inputs are listed in Table 6.7. The acq_ctrl module outputs are listed in Table 6.8. The code for the acq_ctrl module is included in the appendix.

6.3.3. adc. The adc module acquires samples from the analog-to-digital converter. The adc module packages the 12-bit samples into 24-bit words which are output at half the sample rate. The adc module inputs are listed in Table 6.9. The adc module outputs are listed in Table 6.10. The code for the adc module is included in the appendix.

6.3.4. baud_en. The baud_en module is a baud rate generator. The baud_en module generates an enable signal that determines the baud rate of the tx and rx modules. The baud_en module inputs are listed in Table 6.11. The baud_en module outputs are listed in Table 6.12. The code for the baud_en module is included in the appendix.

6.3.5. buff. The buff module is a 12 kB buffer in which acquired samples are stored when the acquisition controller is in single acquisition mode. The buff module inputs are listed in Table 6.13. The buff module outputs are listed in Table 6.14. The code for the buff module is included in the appendix.

6.3.6. cf_rset. The cf_rset module resets the CompactFlash card by driving its –RESET pin low. The cf_rset module checks the card status register to verify that the card has reset. The cf_rset module is controlled by the pio_ctrl module. The cf_rset module controls the pio module. The cf_rset module inputs are listed in Table 6.15. The cf_rset module outputs are listed in Table 6.16. The code for the cf_rset module is included in the appendix.

6.3.7. clk_en. The clk_en module generates enable signals which are used with the 125 MHz system clock to clock modules at 62.5, 31.25, and 7.8125 MHz. The clk_en module inputs are listed in Table 6.17. The clk_en module outputs are listed in Table 6.18. The code for the clk_en module is included in the appendix.

6.3.8. countr. The countr module is a 20 bit frequency counter that is used in the horizontal calibration. The horizontal calibration is discussed in section 2.4. The counter module counts the number of rising edges that occur at its input within a specified number of clock cycles. The countr module inputs are listed in Table 6.19. The countr module outputs are listed in Table 6.20. The code for the countr module is included in the appendix.

6.3.9. delay. The delay module sequentially delays the impulse generator trigger signals generated by the pulse module. The delay increment is read from the delay increment register in the reg_bank module. The delay module inputs are listed in Table 6.21. The delay module outputs are listed in Table 6.22. The code for the delay module is included in the appendix.

6.3.10. pio. The pio module writes to and reads from the CompactFlash card in the PIO mode determined and specified by the pio_mode module. The pio module is controlled by the cf_rset and pio_rw modules. The pio module inputs are listed in Table 6.23. The pio module outputs are listed in Table 6.24. The pio module inputs/outputs are listed in Table 6.25. The code for the pio module is included in the appendix.

6.3.11. pio_buff. The pio_buff module is a pair of 768 B buffers that transfer data from the ram_ctrl module to the pio_rw module. The 24-bit words from the ram_ctrl module are packaged into 16-bit words before being transferred to the pio_rw module. The pio_buff module inputs are listed in Table 6.26. The pio_buff module outputs are listed in Table 6.27. The code for the pio_buff module is included in the appendix.

6.3.12. pio_ctrl. The pio_ctrl module controls the cf_rset, pio_mode, and pio_rw modules. The pio_ctrl module inputs are listed in Table 6.28. The pio_ctrl module outputs are listed in Table 6.29. The pio_ctrl module inputs/outputs are listed in Table 6.30. The code for the pio_ctrl module is included in the appendix.

6.3.13. pio_init. The pio_init module initializes the CompactFlash card for the writing and reading of data in PIO mode. The pio_init module is controlled by the pio_rw module. The pio_init module inputs are listed in Table 6.31. The pio_init module outputs are listed in Table 6.32. The code for the pio_init module is included in the appendix.

6.3.14. pio_mode. The pio_mode module determines the maximum PIO mode supported by the CompactFlash card. The pio_mode module is controlled by the pio_ctrl module. The pio_mode module controls the pio_rw module. The pio_mode module inputs are listed in Table 6.33. The pio_mode module outputs are listed in Table 6.34. The code for the pio_mode module is included in the appendix.

6.3.15. pio_rw. The pio_rw module controls the writing and reading of data to and from the CompactFlash card in PIO mode. The pio_rw module controls the pio and pio_init modules. The pio_rw module is controlled by the pio_mode and pio_ctrl modules. The pio_rw module inputs are listed in Table 6.35. The pio_rw module outputs are listed in Table 6.36. The code for the pio_rw module is included in the appendix.

6.3.16. pulse. The pulse module generates signals to trigger the impulse generator. The signals are sequentially delayed by the delay module before triggering the impulse generator. The pulse module inputs are listed in Table 6.37. The pulse module outputs are listed in Table 6.38. The code for the pulse module is included in the appendix.

6.3.17. ram. The ram module controls the writing and reading of data to and from the SDRAM. The ram module also controls refreshing of the SDRAM. The ram module is controlled by the ram_ctrl module. The ram module inputs are listed in Table 6.39. The ram module outputs are listed in Table 6.40. The ram module inputs/outputs are listed in Table 6.41. The code for the ram module is included in the appendix.

6.3.18. ram_ctrl. The ram_ctrl module controls the transfer of data to and from the ram module. The ram_ctrl module contains a 1.5 kB FIFO from which it transfers data to the ram module. The ram_ctrl module transfers data from the ram module to the pio_buff module. The ram_ctrl module inputs are listed in Table 6.42. The ram_ctrl module outputs are listed in Table 6.43. The ram_ctrl module inputs/outputs are listed in Table 6.44. The code for the ram_ctrl module is included in the appendix.

6.3.19. ram_init. The ram_init module initializes the SDRAM at power on. The ram_init module inputs are listed in Table 6.45. The ram_init module outputs are listed in Table 6.46. The code for the ram_init module is included in the appendix.

6.3.20. reg_bank. The reg_bank module stores acquisition parameters which are read by other modules. The reg_bank module is controlled by the uart_ctrl and rom_ctrl modules. The reg_bank module inputs are listed in Table 6.47. The reg_bank module outputs are listed in Table 6.48. The code for the reg_bank module is included in the appendix.

6.3.21. reg_mux. The reg_mux module switches control of the reg_bank module between the uart_ctrl and rom_ctrl modules. The reg_mux module inputs are listed in Table 6.49. The reg_mux module outputs are listed in Table 6.50. The code for the reg_mux module is included in the appendix.

6.3.22. reg_rd_rom_wr. The reg_rd_rom_wr module transfers acquisition parameters from the reg_bank module to the rom module. The reg_rd_rom_wr module is controlled by the rom_ctrl module. The reg_rd_rom_wr module inputs are listed in Table 6.51. The reg_rd_rom_wr module outputs are listed in Table 6.52. The code for the reg_rd_rom_wr module is included in the appendix.

6.3.23. rom. The rom module controls the writing and reading of acquisition parameters to and from the flash ROM. The rom module inputs are listed in Table 6.53. The rom module outputs are listed in Table 6.54. The code for the rom module is included in the appendix.

6.3.24. rom_ctrl. The rom_ctrl module controls the transfer of acquisition parameters between the reg_bank and rom modules by controlling the rom, reg_rd_rom_wr, and rom_rd_reg_wr modules. The rom_ctrl module inputs are listed in Table 6.55. The rom_ctrl outputs are listed in Table 6.56. The code for the rom_ctrl module is included in the appendix.

6.3.25. rom_init. The rom_init module initializes the flash ROM for read and write operations. The rom_init module is controlled by the rom module. The rom_init module inputs are listed in Table 6.57. The rom_init module outputs are listed in Table 6.58. The code for the rom_init module is included in the appendix.

6.3.26. rom_rd_reg_wr. The rom_rd_reg_wr module transfers acquisition parameters from the rom module to the reg_bank module. The rom_rd_reg_wr module is controlled by the rom_ctrl module. The rom_rd_reg_wr module inputs are listed in

Table 6.59. The rom_rd_reg_wr module outputs are listed in Table 6.60. The code for the rom_rd_reg_wr module is included in the appendix.

6.3.27. rx. The rx module receives serial data from the USB-RS232 converter. The rx module is controlled by the uart_ctrl module. The rx module inputs are listed in Table 6.61. The rx module outputs are listed in Table 6.62. The code for the rx module is included in the appendix.

6.3.28. tdr. The tdr module is a top-level module containing and connecting all other modules. The tdr module inputs are listed in Table 6.63. The tdr module outputs are listed in Table 6.64. The tdr module inputs/outputs are listed in Table 6.65. The code for the tdr module is included in the appendix.

6.3.29. tx. The tx module transmits serial data to the USB-RS232 converter. The tx module is controlled by the uart_ctrl module. The tx module inputs are listed in Table 6.66. The tx module outputs are listed in Table 6.67. The code for the tx module is included in the appendix.

6.3.30. uart. The uart module contains the tx, rx, and baud_en modules. The uart module inputs are listed in Table 6.68. The uart module outputs are listed in Table 6.69. The code for the uart module is included in the appendix.

6.3.31. uart_ctrl. The uart_ctrl module controls the transmission and reception of serial data to and from the USB-RS232 converter by controlling the tx and rx modules. The uart_ctrl module inputs are listed in Table 6.70. The uart_ctrl module outputs are listed in Table 6.71. The code for the uart_ctrl module is included in the appendix.

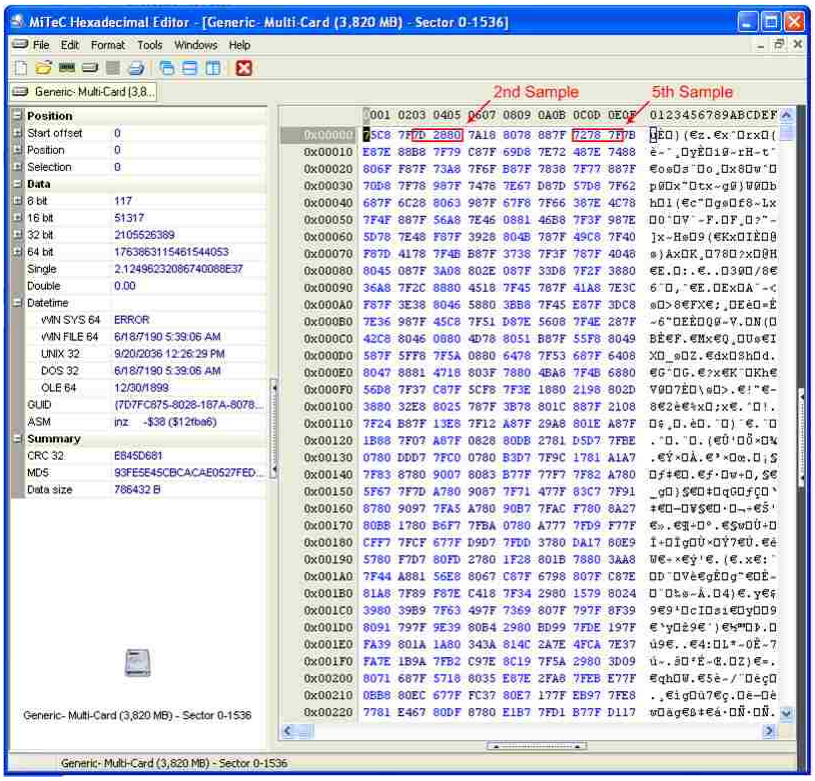

Figure 6.1. HEXEdit Hex Editor

Table 6.1. Control Tab Controls

Control	Acquisition Mode	Values
Port	Single, Multiple	FTDI Virtual COM Port
Baud Rate	Single, Multiple	8000000
Storage Mode	Single, Multiple	RAM, CompactFlash
Sample Rate (GS/s)	Single, Multiple	12.5, 25, 50, 100
Pulse Period (ns)	Single, Multiple	96, 112, 128, … , 4096
Delay (ns)	Single, Multiple	0, 8, 16, … , 1944
Record Length (ns)	Single, Multiple	16, 32, 48, … , 4096
Acquisitions Off	Multiple	1, 2, 3, … , 256
Acquisitions On	Multiple	1, 2, 3, … , 256
Starting Sector	Multiple	0, 1, 2, … , 268435000
Sectors	Multiple	192, 384, 576, … , 393216
Countdown (s)	Single, Multiple	0, 0.1, 0.2, … , Inf
Smoothing Samples	Single, Multiple	1, 3, 5, … , Inf
Smoothing Iterations	Single, Multiple	1, 2, 3, … , 2147483647
Averages	Single	0, 1, 2, … , 65535
Rise Time (ps)	Single, Multiple	20, 21, 22, … , 200
Units	Single, Multiple	Voltage, Reflection Coefficient, Impedance
Response Type	Single, Multiple	Impulse, Step
Save File Path	Single	Valid File Path
Save	Single	Enabled, Disabled
Normalize	Single, Multiple	Enabled, Disabled
De-embed	Single, Multiple	Enabled, Disabled
Load ROM	Single, Multiple	Enabled, Disabled
Smooth	Single, Multiple	Enabled, Disabled
Acquire Single	Single, Multiple	Enabled, Disabled
Acquire Continuous	Single, Multiple	Enabled, Disabled
Armed	Multiple	Enabled, Disabled
Calibrate	Single, Multiple	Enabled, Disabled
Window	Single, Multiple	Enabled, Disabled
Temperature Control	Single, Multiple	Enabled, Disabled
Cursor 1	Single	Enabled, Disabled
Cursor 2	Single	Enabled, Disabled

Table 6.2. Control Tab Indicators

Indicator	**Acquisition Mode**
Temperature Good	Single, Multiple
Loading Registers	Single, Multiple
Acquiring Data	Single, Multiple
Processing Data	Single
Saving Data	Single
Loading ROM	Single, Multiple
Progress	Single
Acquisition Number	Single
Elapsed Time (s)	Single, Multiple
Sample Period (s)	Single, Multiple
Samples Per Acquisition	Single, Multiple
Pulses Per Acquisition	Single, Multiple
Acquisition Time (s)	Single, Multiple
Acquisition Period (s)	Multiple
Acquisition Frequency (Hz)	Multiple
Acquired Signatures	Multiple
Averaged Acquired Signatures	Multiple
Acquired Data (B)	Multiple
Acquisition Off Time (s)	Multiple
Acquisition On Time (s)	Multiple
Averaged Acquisition Period (s)	Multiple
Averaged Acquisition Frequency (Hz)	Multiple
Measurement Time (s)	Multiple
X1	Single
X2	Single
ΔX	Single
Y1	Single
Y2	Single
ΔY	Single
Excess Reactance	Single
XY Graph	Single

Table 6.3. Process Tab Controls

Control	Acquisition Mode	Values
Baseline File Path	Multiple	Valid File Path
Load File Path	Multiple	Valid File Path
Short File Path	Multiple	Valid File Path
Data File Path	Multiple	Valid File Path
Save Path	Multiple	Valid File Path
Line Width	Multiple	0.5, 1.0, 1.5, 2.0
Line Color	Multiple	32 Bit Color
Contour Levels	Multiple	10, 11, 12, … , 100
Format	Multiple	ai, bmp, emf, eps, fig, jpg, m, pbm, pcx, pdf, pgm, png, ppm, tif
Process	Multiple	Enabled, Disabled
Difference	Multiple	Enabled, Disabled
Linear Plots	Multiple	Enabled, Disabled
Contour Plots	Multiple	Enabled, Disabled
Grid On	Multiple	Enabled, Disabled

Table 6.4. Process Tab Indicators

Indicator	Acquisition Mode
Progress	Multiple
Plot Number	Multiple
Elapsed Time (s)	Multiple

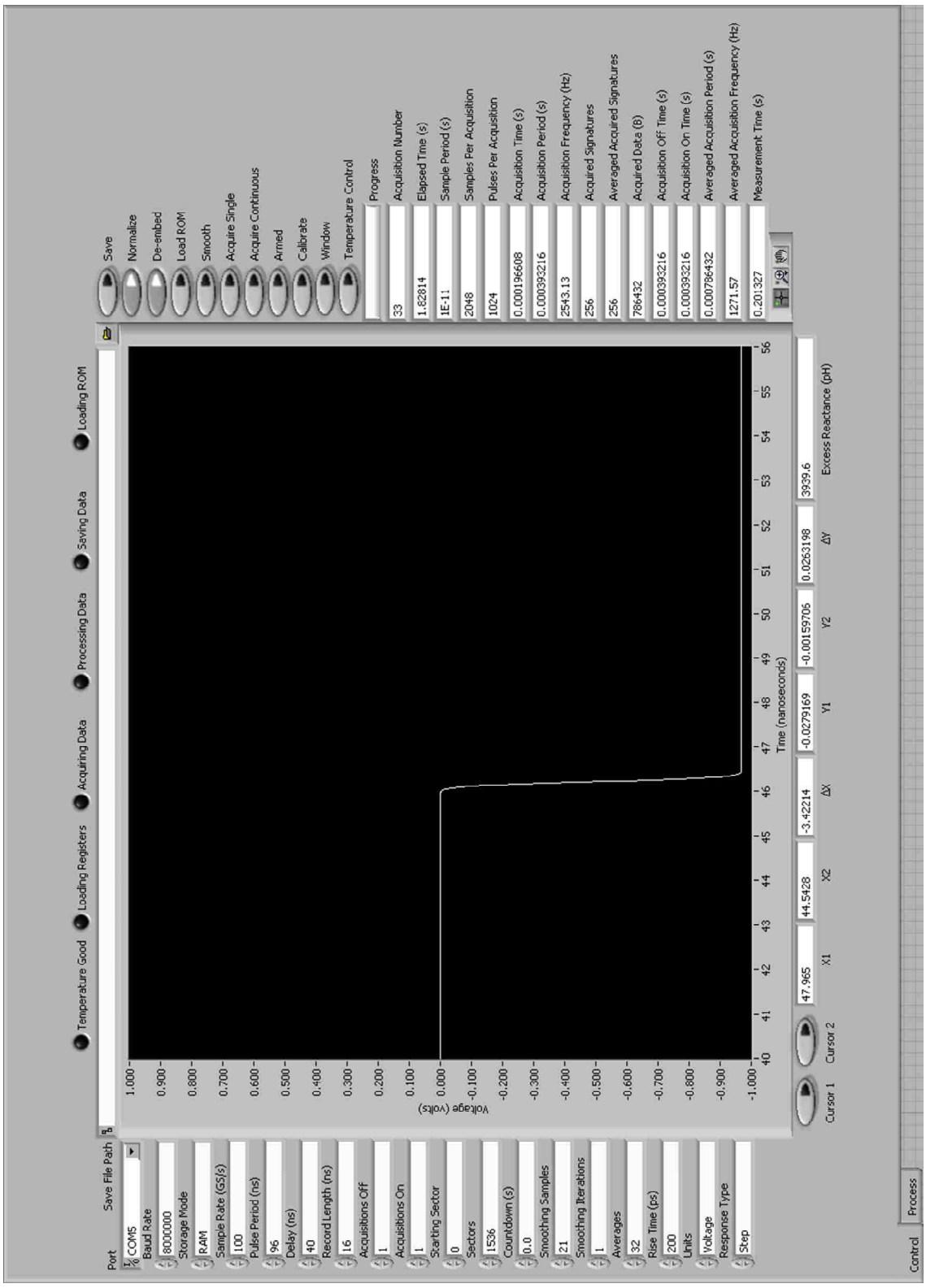

Figure 6.2. Control Tab

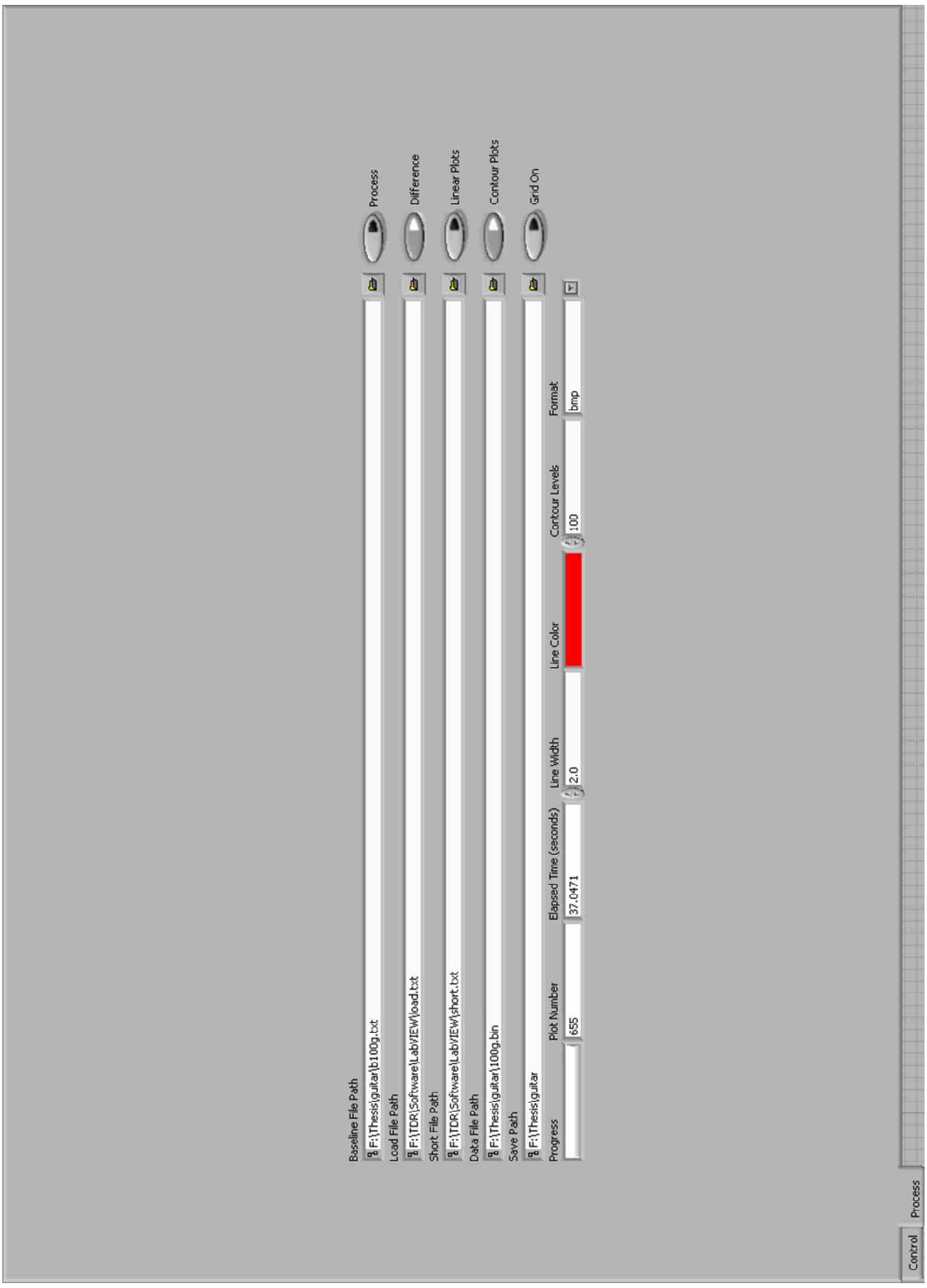

Figure 6.3. Process Tab

Table 6.5. Inputs of acq Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	delay module
trgr	1 bit	pulse module
dlay	7 bits	reg_bank module
wdth	8 bits	reg_bank module
sampls	8 bits	reg_bank module
acq_on	8 bits	reg_bank module
acq_off	8 bits	reg_bank module
steps	10 bits	delay module

Table 6.6. Outputs of acq Module

Output	Width	Destination
wr_en	1 bit	acq_ctrl module
acq_en	1 bit	acq_ctrl module

Table 6.7. Inputs of acq_ctrl Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
clk_en_4	1 bit	clk_en module
invrt	1 bit	reg_bank module
strt	1 bit	reg_bank module
buf_bsy	1 bit	buff module
ram_bsy	1 bit	ram_ctrl module
atrg	1 bit	FPGA input
armd	1 bit	reg_bank module
pio_bsy	1 bit	pio_ctrl module
mode	3 bits	reg_bank module
step	2 bits	reg_bank module
wdth	8 bits	reg_bank module
sampls	8 bits	reg_bank module
adc_dlay	8 bits	reg_bank module
acq_on	8 bits	reg_bank module
acq_off	8 bits	reg_bank module
dval	10 bits	reg_bank module
adc_data	12 bits	FPGA input

Table 6.8. Outputs of acq_ctrl Module

Output	Width	Destination
wr_en	1 bit	buff and ram_ctrl modules
puls	1 bit	FPGA output
len	1 bit	FPGA output
dlay	10 bits	FPGA output
ram_data	24 bits	buff and ram_ctrl modules
bsy	1 bit	reg_bank module
buf_strt	1 bit	buff module
ram_strt	1 bit	ram_ctrl module
disarm	1 bit	reg_bank module

Table 6.9. Inputs of adc Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
clk_en_4	1 bit	clk_en module
mode	1 bit	reg_bank module
dlay	1 bit	reg_bank module
din	12 bit	FPGA input

Table 6.10. Outputs of adc Module

Output	Width	Destination
dout	24	buff and ram_ctrl modules

Table 6.11. Inputs of baud_en Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_4	1 bit	clk_en module
rate	1 bit	tdr module

Table 6.12. Outputs of baud_en Module

Output	Width	Destination
baud_en_1	1 bit	rx module
baud_en_16	1 bit	tx module

Table 6.13. Inputs of buff Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
rd_en	1 bit	uart_ctrl module
strt	1 bit	acq_ctrl module
wr_en	1 bit	acq_ctrl module
wrd_num	12 bits	reg_bank module
wr_data	24 bits	acq_ctrl module

Table 6.14. Outputs of buff Module

Output	Width	Destination
bsy	1 bit	acq_ctrl module
rd_data	24 bits	uart_ctrl module

Table 6.15. Inputs of cf_rset Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	pio_ctrl module
pio_bsy	1 bit	pio module
pio_rd_data	8 bits	pio module

Table 6.16. Outputs of cf_rset Module

Output	Width	Destination
pio_wr	1 bit	pio_ctrl module
nreset	1 bit	FPGA output
pio_adr	3 bits	pio_ctrl module
bsy	1 bit	pio_ctrl module
pio_strt	1 bit	pio_ctrl module

Table 6.17. Inputs of clk_en Module

Input	Width	Source
clk	1 bit	clkbuf module

Table 6.18. Outputs of clk_en Module

Output	Width	Destination
clk_en_2	1 bit	all modules except baud_en, countr, rom, rom_init, rx, tdr, tx, and uart modules
clk_en_4	1 bit	acq_ctrl, adc, bauden, and uart modules
clk_en_16	1 bit	countr, rom, rom_ctrl, and rom_init modules

Table 6.19. Inputs of countr Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_16	1 bit	clk_en module
fin	1 bit	FPGA input
strt	1 bit	reg_bank module

Table 6.20. Outputs of countr Module

Output	Width	Destination
bsy	1 bit	reg_bank module
dout	20 bits	reg_bank module

Table 6.21. Inputs of delay Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
en	1 bit	reg_bank module
trgr	1 bit	pulse module
step	2 bits	reg_bank module
dval	10 bits	reg_bank module

Table 6.22. Outputs of delay Module

Output	Width	Destination
len	1 bit	FPGA output
acq_strt	1 bit	acq and acq_ctrl modules
dlay	10 bits	FPGA output
dcnt	10 bits	acq module

Table 6.23. Inputs of pio Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	pio_ctrl module
wr	1 bit	pio_ctrl module
iordy	1 bit	FPGA input
mode	3 bits	pio_mode module
adr_in	3 bits	pio_ctrl module
wr_data	16 bits	pio_rw module

Table 6.24. Outputs of pio Module

Output	Width	Destination
bsy	1 bit	cf_rset and pio_rw modules
niord	1 bit	FPGA output
niowr	1 bit	FPGA output
adr_out	3 bits	FPGA output
rd_data	8 bits	cf_rset and pio_rw modules

Table 6.25. Inputs/Outputs of pio Module

Input/Output	Width	Source/Destination
cf_data	16 bits	FPGA input/output

Table 6.26. Inputs of pio_buff Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
wr_in	1 bit	ram_ctrl module
ful_in	1 bit	ram_ctrl module
nxt_in	1 bit	pio_rw module
din	24 bits	ram_ctrl module

Table 6.27. Outputs of pio_buff Module

Output	Width	Destination
wr_out	1 bit	pio_rw module
ful_out	1 bit	pio_rw and pio_ctrl modules
nxt_out	1 bit	ram_ctrl module
dout	16 bits	pio_rw module

Table 6.28. Inputs of pio_ctrl Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
iordy	1 bit	FPGA input
wr_en	1 bit	ram_ctrl module
buf_ful	1 bit	ram_ctrl module
ram_bsy	1 bit	ram_ctrl module
wr_data	24 bits	ram_ctrl module
lba_strt	28 bits	reg_bank module

Table 6.29. Outputs of pio_ctrl Module

Output	Width	Destination
ncs0	1 bit	FPGA output
ncs1	1 bit	FPGA output
ndmack	1 bit	FPGA output
nreset	1 bit	FPGA output
niord	1 bit	FPGA output
niowr	1 bit	FPGA output
buf_nxt	1 bit	ram_ctrl module
cf_adr	3 bits	FPGA output
mode	3 bits	reg_bank module
bsy	1 bit	acq_ctrl module

Table 6.30. Inputs/Outputs of pio_ctrl Module

Input/Output	Width	Source/Destination
cf_data	16 bits	FPGA input/output

Table 6.31. Inputs of pio_init Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	pio_rw module
pio_bsy	1 bit	pio module
num_sec	8 bits	pio_ctrl module
lba	28 bits	pio_ctrl module

Table 6.32. Outputs of pio_init Module

Output	Width	Destination
pio_wr	1 bit	pio_rw module
pio_strt	1 bit	pio_rw module
bsy	1 bit	pio_rw module
adr	3 bits	pio_rw module
data	8 bits	pio_rw module

Table 6.33. Inputs of pio_mode Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	pio_ctrl module
rw_bsy	1 bit	pio_rw module
wrd	3 bits	pio_rw module

Table 6.34. Outputs of pio_mode Module

Output	Width	Destination
wr	1 bit	pio_ctrl module
rw_strt	1 bit	pio_ctrl module
bsy	1 bit	pio_ctrl module
mode	3 bits	reg_bank module
wrd_num	9 bits	pio_rw module

Table 6.35. Inputs of pio_rw Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	pio_ctrl module
wr	1 bit	pio_ctrl module
wr_en	1 bit	pio_buff module
pio_bsy	1 bit	pio module
buf_ful	1 bit	pio_buff module
num_sec	8 bits	pio_ctrl module
pio_rd_data	8 bits	pio module
wrd_num	9 bits	pio_mode module
wr_data	16 bits	pio_buff module
lba	28 bits	pio_ctrl module

Table 6.36. Outputs of pio_rw Module

Output	Width	Destination
pio_strt	1 bit	pio_ctrl module
pio_wr	1 bit	pio_ctrl module
pio_adr	3 bits	pio_ctrl module
pio_wr_data	16 bits	pio module
buf_nxt	1 bit	pio_buff module
bsy	1 bit	pio_mode and pio_ctrl modules
word	3 bits	pio_mode module

Table 6.37. Inputs of pulse Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
en	1 bit	reg_bank module
invrt	1 bit	reg_bank module
wdth	8 bits	reg_bank module

Table 6.38. Outputs of pulse Module

Output	Width	Destination
trgr	1 bit	delay and acq modules
puls	1 bit	FPGA output

Table 6.39. Inputs of ram Module

Input	Width	Source
clk_in	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
wr	1 bit	ram_ctrl module
strt	1 bit	ram_ctrl module
wr_data	24 bits	ram_ctrl module
adr_in	26 bits	ram_ctrl module

Table 6.40. Outputs of ram Module

Output	Width	Destination
dqm	1 bit	FPGA output
cke	1 bit	FPGA output
ncs	1 bit	FPGA output
nras	1 bit	FPGA output
ncas	1 bit	FPGA output
nwe	1 bit	FPGA output
adr_out	13 bits	FPGA output
clk_out	1 bit	FPGA output
rd_en	1 bit	ram_ctrl module
wr_en	1 bit	pio_ctrl module
bsy	1 bit	ram_ctrl module
badr	2 bits	FPGA output
rd_data	24 bits	pio_ctrl module

Table 6.41. Inputs/Outputs of ram Module

Input/Output	Width	Source/Destination
dq	24 bits	FPGA input/output

Table 6.42. Inputs of ram_ctrl Module

Input	Width	Source
clk_in	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	acq_ctrl module
wr_en	1 bit	acq_ctrl module
buf_nxt	1 bit	pio_ctrl module
blk_num	11 bits	reg_bank module
wr_data	24 bits	acq_ctrl module

Table 6.43. Outputs of ram_ctrl Module

Output	Width	Destination
clk_out	1 bit	FPGA output
dqm	1 bit	FPGA output
cke	1 bit	FPGA output
ncs	1 bit	FPGA output
nras	1 bit	FPGA output
ncas	1 bit	FPGA output
nwe	1 bit	FPGA output
pio_wr_en	1 bit	pio_ctrl module
badr	2 bits	FPGA output
adr_out	13 bits	FPGA output
rd_data	24 bits	pio_ctrl module
bsy	1 bit	acq_ctrl and pio_ctrl modules
buf_ful	1 bit	pio_ctrl module

Table 6.44. Inputs/Outputs of ram_ctrl Module

Input/Output	Width	Source/Destination
dq	24 bits	FPGA input/output

Table 6.45. Inputs of ram_init Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module

Table 6.46. Outputs of ram_init Module

Output	Width	Destination
adr	13 bits	ram module
bsy	1 bit	ram module
cke	1 bit	ram module
ncs	1 bit	ram module
nras	1 bit	ram module
ncas	1 bit	ram module
nwe	1 bit	ram module

Table 6.47. Inputs of reg_bank Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
reg_strt	1 bit	reg_mux module
wr_en	1 bit	reg_mux module
acq_bsy	1 bit	acq_ctrl module
rom_ctrl_bsy	1 bit	rom_ctrl module
disarm	1 bit	acq_ctrl module
tmpgd	1 bit	FPGA input
fbsy	1 bit	countr module
pio_mode	3 bits	pio_ctrl module
addr	5 bits	reg_mux module
wr_data	8 bits	reg_mux module
fval	20 bits	countr module

Table 6.48. Outputs of reg_bank Module

Output	Width	Destination
load	1 bit	rom_ctrl module
fstrt	1 bit	countr module
acq_strt	1 bit	acq_ctrl module
invrt	1 bit	acq_ctrl module
armd	1 bit	acq_ctrl module
tmpen	1 bit	FPGA output
mode	3 bits	acq_ctrl module and FPGA output
step	2 bits	acq_ctrl module
attn	6 bits	none
wdth	8 bits	acq_ctrl module
sampls	8 bits	acq_ctrl module
adc_dlay	8 bits	acq_ctrl module
acq_on	8 bits	acq_ctrl module
acq_off	8 bits	acq_ctrl module
dval	10 bits	acq_ctrl module
blk	11 bits	ram_ctrl module
buf_wrd_num	12 bits	buff and uart_ctrl modules
lba	28 bits	pio_ctrl module
reg_bsy	1 bit	uart_ctrl and rom_ctrl modules
rd_data	8 bits	uart_ctrl and rom_ctrl modules

Table 6.49. Inputs of reg_mux Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
uart_ctrl_reg_strt	1 bit	uart_ctrl module
rom_ctrl_reg_strt	1 bit	rom_ctrl module
rom_ctrl_bsy	1 bit	rom_ctrl module
uart_ctrl_reg_wr	1 bit	uart_ctrl module
rom_ctrl_reg_wr	1 bit	rom_ctrl module
uart_ctrl_reg_addr	5 bits	uart_ctrl module
rom_ctrl_reg_addr	5 bits	rom_ctrl module
uart_ctrl_reg_wr_data	8 bits	uart_ctrl module
rom_ctrl_reg_wr_data	8 bits	rom_ctrl module

Table 6.50. Outputs of reg_mux Module

Output	Width	Destination
reg_strt	1 bit	reg_bank module
reg_wr	1 bit	reg_bank module
reg_addr	5 bits	reg_bank module
reg_wr_data	8 bits	reg_bank module

Table 6.51. Inputs of reg_rd_rom_wr Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	rom_ctrl module
reg_bsy	1 bit	reg_bank module
rom_bsy	1 bit	rom module
reg_rd_data	8 bits	reg_bank module

Table 6.52. Outputs of reg_rd_rom_wr Module

Output	Width	Destination
bsy	1 bit	rom_ctrl module
reg_strt	1 bit	rom_ctrl module
rom_strt	1 bit	rom_ctrl module
rom_wr_data	8 bits	rom module

Table 6.53. Inputs of rom Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_16	1 bit	clk_en_module
strt	1 bit	rom_ctrl module
wr	1 bit	rom_ctrl module
miso	1 bit	FPGA input
addr	5 bits	rom_ctrl module
wr_data	8 bits	reg_rd_rom_wr module

Table 6.54. Outputs of rom Module

Output	Width	Destination
nss	1 bit	FPGA output
mosi	1 bit	FPGA output
sck	1 bit	FPGA output
bsy	1 bit	reg_rd_rom_wr and rom_rd_reg_wr modules
rd_data	8 bits	rom_rd_reg_wr module

Table 6.55. Inputs of rom_ctrl Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
clk_en_16	1 bit	clk_en module
miso	1 bit	FPGA output
load	1 bit	reg_bank module
reg_bsy	1 bit	reg_bank module
reg_rd_data	8 bits	reg_bank module

Table 6.56. Outputs of rom_ctrl Module

Output	Width	Destination
nss	1 bit	FPGA output
sck	1 bit	FPGA output
mosi	1 bit	FPGA output
reg_strt	1 bit	reg_mux module
reg_addr	5 bits	reg_mux module
reg_wr_data	8 bits	reg_mux module
bsy	1 bit	reg_bank and reg_mux modules
reg_wr	1 bit	reg_mux module

Table 6.57. Inputs of rom_init Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_16	1 bit	clk_en module
wr	1 bit	rom_ctrl module
strt	1 bit	rom_ctrl module
sck	1 bit	FPGA output

Table 6.58. Outputs of rom_init Module

Output	Width	Destination
bsy	1 bit	rom module
nss	1 bit	FPGA output
mosi	1 bit	FPGA output

Table 6.59. Inputs of rom_rd_reg_wr Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
strt	1 bit	rom_ctrl module
reg_bsy	1 bit	reg_bank module
rom_bsy	1 bit	rom module
rom_rd_data	8 bits	rom module

Table 6.60. Outputs of rom_rd_reg_wr Module

Output	Width	Destination
bsy	1 bit	rom_ctrl module
reg_strt	1 bit	rom_ctrl module
rom_strt	1 bit	rom_ctrl module
reg_wr_data	8 bits	reg_mux module

Table 6.61. Inputs of rx Module

Input	Width	Source
clk	1 bit	clkbuf module
baud_en_1	1 bit	baud_en module
din	1 bit	FPGA input
rset	1 bit	uart_ctrl module

Table 6.62. Outputs of rx Module

Output	Width	Destination
full	1 bit	uart_ctrl module
dout	8 bits	uart_ctrl module

Table 6.63. Inputs of tdr Module

Input	Width	Source
clk_p	1 bit	FPGA input
clk_n	1 bit	FPGA input
usb_rx	1 bit	FPGA input
iordy	1 bit	FPGA input
miso	1 bit	FPGA input
atrg	1 bit	FPGA input
tmpgd	1 bit	FPGA input
fin	1 bit	FPGA input
adc_data	12 bits	FPGA input

Table 6.64. Outputs of tdr Module

Output	Width	Destination
sck	1 bit	FPGA output
mosi	1 bit	FPGA output
rom_nrset	1 bit	FPGA output
rom_nwp	1 bit	FPGA output
rom_nss	1 bit	FPGA output
adc_nss	1 bit	FPGA output
dac_nss	1 bit	FPGA output
usb_tx	1 bit	FPGA output
ram_clk_out	1 bit	FPGA output
ram_cke	1 bit	FPGA output
ram_ncs	1 bit	FPGA output
ram_nras	1 bit	FPGA output
ram_ncas	1 bit	FPGA output
ram_nwe	1 bit	FPGA output
ram_dqm	1 bit	FPGA output
ncs0	1 bit	FPGA output
ncs1	1 bit	FPGA output
ndmack	1 bit	FPGA output
nreset	1 bit	FPGA output
niord	1 bit	FPGA output
niowr	1 bit	FPGA output
puls	1 bit	FPGA output
len	1 bit	FPGA output
sel	1 bit	FPGA output
tmpen	1 bit	FPGA output
ram_badr	2 bits	FPGA output
cf_adr	3 bits	FPGA output
dlay	10 bits	FPGA output
ram_adr_out	13 bits	FPGA output

Table 6.65. Inputs/Outputs of tdr Module

Input/Output	Width	Source/Destination
cf_data	16 bits	FPGA input/output
ram_dq	24 bits	FPGA input/output

Table 6.66. Inputs of tx Module

Input	Width	Source
clk	1 bit	clkbuf module
baud_en_16	1 bit	baud_en module
strt	1 bit	uart_ctrl module
din	8 bits	uart_ctrl module

Table 6.67. Outputs of tx Module

Output	Width	Destination
bsy	1 bit	uart_ctrl module
dout	1 bit	FPGA output

Table 6.68. Inputs of uart Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_4	1 bit	clk_en module
rate	1 bit	tdr module
rx_din	1 bit	FPGA input
rx_rset	1 bit	uart_ctrl module
tx_strt	1 bit	uart_ctrl module
tx_din	8 bits	uart_ctrl module

Table 6.69. Outputs of uart Module

Output	Width	Destination
rx_full	1 bit	uart_ctrl module
tx_bsy	1 bit	uart_ctrl module
tx_dout	1 bit	FPGA output
rx_dout	8 bits	uart_ctrl module

Table 6.70. Inputs of uart_ctrl Module

Input	Width	Source
clk	1 bit	clkbuf module
clk_en_2	1 bit	clk_en module
rx_full	1 bit	uart module
tx_bsy	1 bit	uart module
reg_bsy	1 bit	reg_bank module
rx_data	8 bits	uart module
rd_data	8 bits	reg_bank module
wrd_num	12 bits	reg_bank module
buf_data	24 bits	buff module

Table 6.71. Outputs of uart_ctrl Module

Output	Width	Destination
reg_strt	1 bit	reg_mux module
buf_rd	1 bit	buff module
wr_en	1 bit	reg_mux module
rx_rset	1 bit	uart module
tx_strt	1 bit	uart module
addr	5 bits	reg_mux module
tx_data	8 bits	uart module
wr_data	8 bits	reg_mux module

7. CONCLUSIONS

The acquisition rate of the TDR was evaluated by modulating the voltage reflected from a short termination with a voltage variable attenuator. The TDR was able to monitor the reflected voltage at modulation frequencies as high as 1 kHz. The acquisition rate of the TDR was further evaluated by modulating the voltage reflected from a wire over ground plane transmission line by vibrating the wire. The TDR was able to monitor the 117 Hz vibration of the wire and the mechanical standing waves were apparent in the step response. The TDR was applied in the monitor of a crack sensor embedded in a bridge column during a shake-table experiment. The TDR was able to monitor the evolution of a crack which formed in the column 0.35 m from the beginning of the sensor. The TDR performs well, however there are several aspects which can be improved. The dynamic range of the TDR can be increased 6 dB by removing the 3 dB damping attenuators at the outputs of the variable-gain amplifier and power limiter. Removing the damping attenuators, however, amplifies the mismatch reflections in the impulse response of the TDR. The mismatch reflections in the impulse response of the TDR can be attenuated by redesigning the impulse generator, variable-gain amplifier, and power limiter to have better input and output impedance matches. The spatial resolution of the TDR can be improved by integrating the impulse generator, variable-gain amplifier, power splitter, and power limiter into a single printed circuit board. The TDR can be transformed into a time-domain transmissometer (TDT) by the insertion of a single-pole, double-throw switch between the power splitter and power limiter.

APPENDIX

```
module acq(
           clk,
           clk_en_2,
           strt,
           trgr,
           wdth,
           dlay,
           sampls,
           acq_on,
           acq_off,
           steps,
           wr_en,
           acq_en
           );

   input clk;
   input clk_en_2;
   input strt;
   input trgr;
   input [6:0] dlay;
   input [7:0] wdth;
   input [7:0] sampls;
   input [7:0] acq_on;
   input [7:0] acq_off;
   input [9:0] steps;

   output reg wr_en;    // synthesis attribute init of wr_en is "0"
   output reg acq_en;   // synthesis attribute init of acq_en is "0"

   reg [5:0] state;     // synthesis attribute init of state is "000001"
   reg [7:0] count;     // synthesis attribute init of count is "00000000"
   reg [7:0] scnt;      // synthesis attribute init of scnt is "00000000"
   reg [7:0] acnt;      // synthesis attribute init of acnt is "00000000"
   reg [9:0] dcnt;      // synthesis attribute init of dcnt is "0000000000"

   wire [7:0] samplz;

   parameter IDLE       = 6'b000001;
   parameter WAIT_LOW   = 6'b000010;
   parameter WAIT_HIGH  = 6'b000100;
   parameter DLAY       = 6'b001000;
   parameter SAMPL      = 6'b010000;
   parameter WAIT       = 6'b100000;

   assign samplz = wdth < sampls ? wdth : sampls;

   always @(posedge clk)
      if (clk_en_2)
         begin
            if (strt)
               begin
                  acq_en <= acnt ? acq_en : ~ acq_en;
                  acnt <= acnt ? acnt - 8'd1 : acq_en ? acq_off : acq_on;
               end
            case (state)   // synthesis full_case
               IDLE:
                  state <= strt & acq_en ? WAIT_LOW : state;
               WAIT_LOW:
                  state <= ~ trgr ? WAIT_HIGH : state;
               WAIT_HIGH:
                  begin
                     count <= {1'd0 , dlay};
                     state <= trgr ? DLAY : state;
                  end
               DLAY:
                  begin
```

```
                    dcnt <= steps;
                    scnt <= samplz;
                    wr_en <= count ? wr_en : 1'b1;
                    state <= count ? state : SAMPL;
                    count <= count ? count - 8'd1 : wdth;
                 end
              SAMPL:
                 begin
                    scnt <= scnt - 8'd1;
                    wr_en <= scnt ? wr_en : 1'b0;
                    state <= count ? state : WAIT;
                    count <= count ? count - 8'd1 : wdth;
                 end
              WAIT:
                 begin
                    scnt <= samplz;
                    dcnt <= count ? dcnt : dcnt - 10'd1;
                    count <= count ? count - 8'd1 : wdth;
                    wr_en <= count ? wr_en : dcnt ? 1'b1 : wr_en;
                    state <= count ? state : dcnt ? SAMPL : IDLE;
                 end
           endcase
        end

endmodule

module acq_ctrl(
                clk,
                clk_en_2,
                clk_en_4,
                invrt,
                mode,
                strt,
                buf_bsy,
                ram_bsy,
                step,
                wdth,
                sampls,
                adc_dlay,
                buf_strt,
                ram_strt,
                wr_en,
                bsy,
                puls,
                len,
                dlay,
                adc_data,
                ram_data,
                acq_on,
                acq_off,
                dval,
                atrg,
                armd,
                disarm,
                pio_bsy
                );

   input clk;
   input clk_en_2;
   input clk_en_4;
   input invrt;
   input strt;
   input buf_bsy;
   input ram_bsy;
   input atrg;
   input armd;
   input pio_bsy;
   input [2:0] mode;
   input [1:0] step;
   input [7:0] wdth;
   input [7:0] sampls;
```

```
input [7:0] adc_dlay;
input [7:0] acq_on;
input [7:0] acq_off;
input [9:0] dval;
input [11:0] adc_data;

output wr_en;
output puls;
output len;
output [9:0] dlay;
output [23:0] ram_data;

output reg bsy;      // synthesis attribute init of bsy is "0"
output reg buf_strt; // synthesis attribute init of buf_strt is "0"
output reg ram_strt; // synthesis attribute init of ram_strt is "0"
output reg disarm;   // synthesis attribute init of disarm is "0"

reg atrg1;           // synthesis attribute init of atrg1 is "0"
reg atrg2;           // synthesis attribute init of atrg2 is "0"
reg trgr;            // synthesis attribute init of trgr is "0"
reg acq_ctrl_wr_en;  // synthesis attribute init of acq_ctrl_wr_en is "0"
reg [6:0] state;     // synthesis attribute init of state is "0000001"

wire acq_en;
wire acq_strt;
wire acq_wr_en;
wire ptrg;
wire [9:0] steps;

parameter IDLE       = 7'b0000001;
parameter ACQ_WAIT   = 7'b0000010;
parameter ACQ_STRT   = 7'b0000100;
parameter BUF_STRT   = 7'b0001000;
parameter RAM_STRT   = 7'b0010000;
parameter BUF_BUSY   = 7'b0100000;
parameter RAM_BUSY   = 7'b1000000;

adc adc(
        .clk(clk),
        .clk_en_2(clk_en_2),
        .clk_en_4(clk_en_4),
        .mode(mode [1]),
        .dlay(adc_dlay [0]),
        .din(adc_data),
        .dout(ram_data)
        );

pulse pulse(
            .clk(clk),
            .clk_en_2(clk_en_2),
            .en(~ mode [2]),
            .invrt(invrt),
            .wdth(wdth),
            .trgr(ptrg),
            .puls(puls)
            );

delay delay(
            .clk(clk),
            .clk_en_2(clk_en_2),
            .en(~ mode [2]),
            .trgr(ptrg),
            .step(step),
            .dval(dval),
            .len(len),
            .acq_strt(acq_strt),
            .dcnt(steps),
            .dlay(dlay)
            );

acq acq(
```

```
            .clk(clk),
            .clk_en_2(clk_en_2),
            .strt(acq_strt),
            .trgr(ptrg),
            .wdth(wdth),
            .dlay(adc_dlay [7:1]),
            .sampls(sampls),
            .acq_on(acq_on),
            .acq_off(acq_off),
            .steps(steps),
            .wr_en(acq_wr_en),
            .acq_en(acq_en)
            );

   assign wr_en = mode [1] ? acq_wr_en : acq_ctrl_wr_en;

   always @(posedge clk)
      if (clk_en_2)
         begin
            disarm <= 1'b0;
            atrg1 <= atrg;
            atrg2 <= atrg1;
            trgr <= armd & atrg2;
            acq_ctrl_wr_en <= ~ acq_ctrl_wr_en;
            case (state)   // synthesis full_case
               IDLE:
                  if (strt | trgr)
                     begin
                        disarm <= trgr ? 1'b1 : disarm;
                        bsy <= 1'b1;
                        state <= ACQ_WAIT;
                     end
               ACQ_WAIT:
                  if (~acq_en)
                     state <= ACQ_STRT;
               ACQ_STRT:
                  if (acq_strt & acq_en)
                     begin
                        buf_strt <= mode [0] ? buf_strt : 1'b1;
                        ram_strt <= mode [0] ? 1'b1 : ram_strt;
                        state <= mode [0] ? RAM_STRT : BUF_STRT;
                     end
               BUF_STRT:
                  if (buf_bsy)
                     begin
                        buf_strt <= 1'b0;
                        state <= BUF_BUSY;
                     end
               RAM_STRT:
                  if (ram_bsy)
                     begin
                        ram_strt <= 1'b0;
                        state <= RAM_BUSY;
                     end
               BUF_BUSY:
                  if (~buf_bsy)
                     begin
                        bsy <= 1'b0;
                        state <= IDLE;
                     end
               RAM_BUSY:
                  if (~(ram_bsy | pio_bsy))
                     begin
                        bsy <= 1'b0;
                        state <= IDLE;
                     end
            endcase
         end

endmodule
```

```
module adc(
           clk,
           clk_en_2,
           clk_en_4,
           mode,
           dlay,
           din,
           dout
           );

   input clk;
   input clk_en_2;
   input clk_en_4;
   input mode;
   input dlay;
   input [11:0] din;

   output [23:0] dout;

   reg [11:0] sync_1;         // synthesis attribute init of sync_1 is "000000000000"
   reg [11:0] sync_2;         // synthesis attribute init of sync_2 is "000000000000"
   reg [11:0] data_1_1;       // synthesis attribute init of data_1_1 is "000000000000"
   reg [11:0] data_2_1;       // synthesis attribute init of data_2_1 is "000000000000"
   reg [11:0] data_1_2;       // synthesis attribute init of data_1_2 is "000000000000"
   reg [11:0] data_2_2;       // synthesis attribute init of data_2_2 is "000000000000"
   reg [11:0] data_1_3;       // synthesis attribute init of data_1_3 is "000000000000"
   reg [23:0] dout_1;         // synthesis attribute init of dout_1 is
"000000000000000000000000"
   reg [23:0] dout_2;         // synthesis attribute init of dout_2 is
"000000000000000000000000"

   assign dout = mode ? dout_1 : dout_2;

   always @(posedge clk)
      begin
         sync_1 <= din;
         data_1_1 <= sync_1;
         data_1_2 <= data_1_1;
         data_1_3 <= data_1_2;
         if (clk_en_2)
            begin
               sync_2 <= din;
               data_2_1 <= sync_2;
               data_2_2 <= data_2_1;
               dout_1 <= dlay ? {data_1_1 , data_1_2} : {data_1_2 , data_1_3};
            end
         if (clk_en_4)
            dout_2 <= {data_2_1 , data_2_2};
      end

endmodule

module baud_en(
               clk,
               clk_en_4,
               rate,
               baud_en_1,
               baud_en_16
               );

   input clk;
   input clk_en_4;
   input rate;             // 0 = 115200 baud
                           // 1 = 8000000 baud
   output baud_en_1;

   output reg baud_en_16;

   reg baud_en_lo;
   reg [4:0] count1;       // synthesis attribute init of count1 is "00000"
   reg [3:0] count2;       // synthesis attribute init of count2 is "0000"
```

```
   wire rset1;
   wire rset2;
   wire baud_en_hi;

   assign rset1 = count1 [4];
   assign rset2 = & count2;
   assign baud_en_hi = 1'b1;
   assign baud_en_1 = rate ? baud_en_hi : baud_en_lo;

   always @(posedge clk)
      begin
         baud_en_lo <= clk_en_4 & rset1;
         baud_en_16 <= baud_en_1 & rset2;
         if (clk_en_4)
            count1 <= rset1 ? 5'd0 : count1 + 5'd1;
         if (baud_en_1)
            count2 <= count2 + 4'd1;
      end

endmodule

module buff(
            clk,
            clk_en_2,
            rd_en,
            strt,
            wr_en,
            wrd_num,
            wr_data,
            bsy,
            rd_data
            );

   input clk;
   input clk_en_2;
   input rd_en;
   input strt;
   input wr_en;
   input [11:0] wrd_num;
   input [23:0] wr_data;

   output reg bsy;            // synthesis attribute init of bsy is "0"
   output reg [23:0] rd_data; // synthesis attribute init of rd_data is
"000000000000000000000000"

   reg [1:0] state;           // synthesis attribute init of state is "01"
   reg [11:0] wd_num;         // synthesis attribute init of wd_num is "000000000000"
   reg [11:0] rd_adr;         // synthesis attribute init of rd_adr is "000000000000"
   reg [11:0] wr_adr;         // synthesis attribute init of wr_adr is "000000000000"
   reg [23:0] bufr [0:4095];  // synthesis attribute ram_style of bufr is block

   wire rd_done;
   wire wr_done;

   parameter IDLE = 2'b01;
   parameter BUSY = 2'b10;

   assign rd_done = rd_adr == wd_num;
   assign wr_done = wr_adr == wd_num;

   always @(posedge clk)
      if (clk_en_2)
         begin
            wd_num <= wrd_num;
            if (rd_en)
               begin
                  rd_adr <= rd_done ? 12'd0 : rd_adr + 12'd1;
                  rd_data <= bufr [rd_adr];
               end
            case (state)   // synthesis full_case
```

```
               IDLE:
                  if (strt)
                     begin
                        bsy <= 1'b1;
                        wr_adr <= 12'd0;
                        state <= BUSY;
                     end
               BUSY:
                  begin
                     wr_adr <= wr_en ? wr_adr + 12'd1 : wr_adr;
                     bufr [wr_adr] <= wr_en ? wr_data : bufr [wr_adr];
                     bsy <= wr_en & wr_done ? 1'b0 : bsy;
                     state <= wr_en & wr_done ? IDLE : state;
                  end
            endcase
         end

endmodule

module cf_rset(
               clk,
               clk_en_2,
               strt,
               pio_bsy,
               pio_rd_data,
               nreset,
               bsy,
               pio_strt,
               pio_wr,
               pio_adr
               );

   input clk;
   input clk_en_2;
   input strt;
   input pio_bsy;
   input [7:0] pio_rd_data;

   output pio_wr;
   output nreset;
   output [2:0] pio_adr;

   output reg bsy;      // synthesis attribute init of bsy is "0"
   output reg pio_strt; // synthesis attribute init of pio_strt is "0"

   reg reset;           // synthesis attribute init of reset is "0"
   reg [2:0] state;     // synthesis attribute init of state is "001"

   wire rset;
   wire done;
   wire nbsy;

   parameter IDLE       = 3'b001;
   parameter STAT_STRT  = 3'b010;
   parameter STAT_BUSY  = 3'b100;
   parameter stat = 3'd7;

   assign rset = reset & ~ nbsy;
   assign done = nreset & nbsy;
   assign nbsy = pio_rd_data == 8'h50;
   assign nreset = ~ reset;
   assign pio_wr = 1'b0;
   assign pio_adr = stat;

   always @(posedge clk)
      if (clk_en_2)
         case (state)   // synthesis full_case
            IDLE:
               if (strt)
                  begin
                     bsy <= 1'b1;
```

```
                  pio_strt <= 1'b1;
                  reset <= 1'b1;
                  state <= STAT_STRT;
               end
            STAT_STRT:
               if (pio_bsy)
                  begin
                  pio_strt <= 1'b0;
                  state <= STAT_BUSY;
               end
            STAT_BUSY:
               if (~pio_bsy)
                  begin
                  bsy <= done ? 1'b0 : bsy;
                  pio_strt <= done ? pio_strt : 1'b1;
                  reset <= rset ? ~ reset : reset;
                  state <= done ? IDLE : STAT_STRT;
               end
         endcase

endmodule

module clk_en(
              clk,
              clk_en_2,
              clk_en_4,
              clk_en_16
              );

   input clk;

   output reg clk_en_2;
   output reg clk_en_4;
   output reg clk_en_16;

   reg [3:0] count;  // synthesis attribute init of count is "0000"

   always @(posedge clk)
      begin
         clk_en_2 <= count [0];
         clk_en_4 <= & count [1:0];
         clk_en_16 <= & count [3:0];
         count <= count + 4'd1;
      end

endmodule

module countr(
              clk,
              clk_en_16,
              fin,
              strt,
              bsy,
              dout
              );

   input clk;
   input clk_en_16;
   input fin;
   input strt;

   output reg bsy;               // synthesis attribute init of bsy is "0"
   output reg [19:0] dout;       // synthesis attribute init of dout is
"00000000000000000000"

   reg [2:0] state;              // synthesis attribute init of state is "001"
   reg [16:0] clk_count;         // synthesis attribute init of clk_count is
"00000000000000000"
   reg [19:0] f_count;           // synthesis attribute init of f_count is
"00000000000000000000"
```

```
   parameter IDLE    = 4'b001;
   parameter RESET   = 4'b010;
   parameter COUNT   = 4'b100;

   always @(posedge clk)
      if (clk_en_16)
         begin
            dout <= f_count;
            case (state)      // synthesis full_case
               IDLE:
                  if (strt)
                  begin
                     bsy <= 1'b1;
                     state <= RESET;
                  end
               RESET:
                  if (f_count == 20'd0)
                  state <= COUNT;
               COUNT:
                  begin
                     clk_count <= clk_count + 17'd1;
                     if (clk_count [16])
                        begin
                           clk_count <= 17'd0;
                           bsy <= 1'b0;
                           state <= IDLE;
                        end
                  end
            endcase
         end

   always @(posedge fin)
      begin
         if (state == RESET)
            f_count <= 20'd0;
         if (state == COUNT)
            f_count <= f_count + 20'd1;
      end

endmodule

module delay(
             clk,
             clk_en_2,
             en,
             trgr,
             step,
             dval,
             len,
             acq_strt,
             dcnt,
             dlay
             );

   input clk;
   input clk_en_2;
   input en;
   input trgr;
   input [1:0] step;
   input [9:0] dval;

   output reg len;         // synthesis attribute init of len is "1"
   output reg acq_strt;    // synthesis attribute init of acq_strt is "0"
   output reg [9:0] dlay;  // synthesis attribute init of dlay is "0000000000"
   output reg [9:0] dcnt;  // synthesis attribute init of dcnt is "1111111111"

   reg rset;               // synthesis attribute init of rset is "0"
   reg inc;                // synthesis attribute init of inc is "1"
   reg [4:0] state;        // synthesis attribute init of state is "00001"
   reg [9:0] dnum;         // synthesis attribute init of dnum is "0000000000"
   reg [9:0] count;        // synthesis attribute init of count is "0000000000"
```

```
reg [9:0] dstp;         // synthesis attribute init of dstp is "0000000001"
reg [9:0] stp;
reg [9:0] cnt;

parameter WAIT_LOW   = 5'b00001;
parameter DLAY_LOAD  = 5'b00010;
parameter WAIT_HIGH  = 5'b00100;
parameter WAIT_COMP  = 5'b01000;
parameter CHCK_RSET  = 5'b10000;

always @(step)
   case (step)
      2'd0:
         begin
            stp = 10'd1;      // 1024 pulses
            cnt = 10'd1023;   // 1023 steps
         end
      2'd1:
         begin
            stp = 10'd2;      // 512 pulses
            cnt = 10'd511;    // 511 steps
         end
      2'd2:
         begin
            stp = 10'd4;      // 256 pulses
            cnt = 10'd255;    // 255 steps
         end
      2'd3:
         begin
            stp = 10'd8;      // 128 pulses
            cnt = 10'd127;    // 127 steps
         end
   endcase

always @(posedge clk)
   if (clk_en_2)
      begin
         acq_strt <= 1'b0;
         dlay <= en ? dnum : dval;
         case (state)   // synthesis full_case
            WAIT_LOW:
               if (~trgr)
                  begin
                     len <= 1'b0;
                     state <= DLAY_LOAD;
                  end
            DLAY_LOAD:
               begin
                  len <= 1'b1;
                  dnum <= inc ? dnum + dstp : dnum - dstp;
                  state <= WAIT_HIGH;
               end
            WAIT_HIGH:
               if (trgr)
                  begin
                     count <= count + 10'd1;
                     state <= WAIT_COMP;
                  end
            WAIT_COMP:
               begin
                  rset <= count == dcnt;
                  state <= CHCK_RSET;
               end
            CHCK_RSET:
               begin
                  count <= rset ? 10'd0 : count;
                  inc <= rset ? ~ inc : inc;
                  acq_strt <= rset & ~ inc ? 1'b1 : acq_strt;
                  dstp <= rset & ~ inc ? stp : dstp;
                  dcnt <= rset & ~ inc ? cnt : dcnt;
                  state <= WAIT_LOW;
```

```
                end
             endcase
          end

endmodule

module pio(
           clk,
           clk_en_2,
           strt,
           wr,
           iordy,
           mode,
           adr_in,
           wr_data,
           cf_data,
           bsy,
           niord,
           niowr,
           adr_out,
           rd_data
           );

   input clk;
   input clk_en_2;
   input strt;
   input wr;
   input iordy;
   input [2:0] mode;
   input [2:0] adr_in;
   input [15:0] wr_data;

   inout [15:0] cf_data;

   output reg bsy;            // synthesis attribute init of bsy is "0"
   output reg niord;          // synthesis attribute init of niord is "1"
   output reg niowr;          // synthesis attribute init of niowr is "1"
   output reg [2:0] adr_out;  // synthesis attribute init of adr_out is "000"
   output reg [7:0] rd_data;  // synthesis attribute init of rd_data is "00000000"

   reg [3:0] state;           // synthesis attribute init of state is "0001"
   reg [4:0] count;           // synthesis attribute init of count is "00000"

   parameter IDLE    =  4'b0001;
   parameter START   =  4'b0010;
   parameter RW      =  4'b0100;
   parameter STOP    =  4'b1000;

   assign cf_data = wr ? wr_data : 16'hzzzz;

   always @(posedge clk)
      if (clk_en_2)
         case (state)   // synthesis full_case
            IDLE:
               begin
                  adr_out <= adr_in;
                  if (strt)
                     begin
                        bsy <= 1'b1;
                        state <= START;
                     end
                  case (mode) // synthesis full_case
                     3'd0:
                        count <= 5'd4; // ceiling[(70+1)/16]-1
                     3'd1:
                        count <= 5'd3; // ceiling[(50+1)/16]-1
                     3'd2:
                        count <= 5'd1; // ceiling[(30+1)/16]-1
                     3'd3:
                        count <= 5'd1; // ceiling[(30+1)/16]-1
                     3'd4:
```

```
            count <= 5'd1; // ceiling[(25+1)/16]-1
         3'd5:
            count <= 5'd0; // ceiling[(15+1)/16]-1
         3'd6:
            count <= 5'd0; // ceiling[(10+1)/16]-1
      endcase
   end
START:
   begin
      count <= count - 5'd1;
      if (count == 5'd0)
         begin
            niord <= wr ? 1'b1 : 1'b0;
            niowr <= wr ? 1'b0 : 1'b1;
            state <= RW;
            case (mode) // synthesis full_case
               3'd0:
                  count <= 5'd18;   // ceiling[(290+1)/16]-1
               3'd1:
                  count <= 5'd18;   // ceiling[(290+1)/16]-1
               3'd2:
                  count <= 5'd18;   // ceiling[(290+1)/16]-1
               3'd3:
                  count <= 5'd5;    // ceiling[(80+1)/16]-1
               3'd4:
                  count <= 5'd4;    // ceiling[(70+1)/16]-1
               3'd5:
                  count <= 5'd4;    // ceiling[(65+1)/16]-1
               3'd6:
                  count <= 5'd3;    // ceiling[(55+1)/16]-1
            endcase
         end
   end
RW:
   if (count == 5'd0)
      begin
         if (iordy)
            begin
               state <= STOP;
               if (wr)
                  niowr <= 1'b1;
               else
                  begin
                     niord <= 1'b1;
                     rd_data <= cf_data [7:0];
                  end
               case (mode) // synthesis full_case
                  3'd0:
                     count <= 5'd14;   // ceiling[(600-70-290+1)/16]-2
                  3'd1:
                     count <= 5'd1;    // ceiling[(383-50-290+1)/16]-2
                  3'd2:
                     count <= 5'd0;    // ceiling[(240-30-290+1)/16]-2
                  3'd3:
                     count <= 5'd3;    // ceiling[(180-30-80+1)/16]-2
                  3'd4:
                     count <= 5'd0;    // ceiling[(120-25-70+1)/16]-2
                  3'd5:
                     count <= 5'd0;    // ceiling[(100-15-65+1)/16]-2
                  3'd6:
                     count <= 5'd0;    // ceiling[(80-10-55+1)/16]-2
               endcase
            end
      end
   else
      count <= count - 5'd1;
STOP:
   begin
      count <= count - 5'd1;
      if (count == 5'd0)
         begin
```

```
                        bsy <= 1'b0;
                        state <= IDLE;
                     end
               end
         endcase

endmodule

module pio_buff(
                clk,
                clk_en_2,
                wr_in,
                wr_out,
                din,
                dout,
                ful_in,
                ful_out,
                nxt_in,
                nxt_out
                );

   input clk;
   input clk_en_2;
   input wr_in;
   input ful_in;
   input nxt_in;
   input [23:0] din;

   output reg wr_out;         // synthesis attribute init of wr_out is "0"
   output reg ful_out;        // synthesis attribute init of ful_out is "0"
   output reg nxt_out;        // synthesis attribute init of nxt_out is "0"
   output reg [15:0] dout;    // synthesis attribute init of dout is "0000000000000000"

   reg buf_sel;               // synthesis attribute init of buf_sel is "0"
   reg [2:0] wrd;             // synthesis attribute init of wrd is "001"
   reg [7:0] rd_adr;          // synthesis attribute init of rd_adr is "00000000"
   reg [7:0] wr_adr;          // synthesis attribute init of wr_adr is "00000000"
   reg [8:0] rd_cnt;          // synthesis attribute init of rd_cnt is "000000000"
   reg [8:0] wr_cnt;          // synthesis attribute init of wr_cnt is "000000000"
   reg [8:0] state;           // synthesis attribute init of state is "000000001"
   reg [23:0] data;           // synthesis attribute init of data is
"000000000000000000000000"
   reg [23:0] data_1;         // synthesis attribute init of data_1 is
"000000000000000000000000"
   reg [23:0] data_2;         // synthesis attribute init of data_2 is
"000000000000000000000000"
   reg [23:0] buf_0 [0:255];  // synthesis attribute ram_style of buf_0 is block
   reg [23:0] buf_1 [0:255];  // synthesis attribute ram_style of buf_1 is block

   parameter WAIT_BUFF  = 9'b000000001;
   parameter READ_INIT  = 9'b000000010;
   parameter READ_1     = 9'b000000100;
   parameter READ_2     = 9'b000001000;
   parameter READ_3     = 9'b000010000;
   parameter WRITE_INIT = 9'b000100000;
   parameter WRITE      = 9'b001000000;
   parameter BUF_FUL    = 9'b010000000;
   parameter WRITE_DONE = 9'b100000000;
   parameter wrd_1      = 3'b001;
   parameter wrd_2      = 3'b010;
   parameter wrd_3      = 3'b100;

   always @(posedge clk)
      if (clk_en_2)
         begin
            data <= buf_sel ? buf_1 [rd_adr] : buf_0 [rd_adr];
            if (wr_in)
               begin
                  wr_adr <= wr_adr + 8'd1;
                  buf_0 [wr_adr] <= buf_sel ? din : buf_0 [wr_adr];
                  buf_1 [wr_adr] <= buf_sel ? buf_1 [wr_adr] : din;
```

```
      end
   case (state)   // synthesis full_case
      WAIT_BUFF:
         begin
            buf_sel <= ful_in ? ~ buf_sel : buf_sel;
            nxt_out <= ful_in ? 1'b1 : nxt_out;
            state <= ful_in ? READ_INIT : state;
         end
      READ_INIT:
         begin
            nxt_out <= 1'b0;
            rd_cnt <= rd_cnt [8] ? 9'd0 : rd_cnt;
            state <= rd_cnt [8] ? WAIT_BUFF : READ_1;
         end
      READ_1:
         begin
            rd_cnt <= rd_cnt + 9'd1;
            rd_adr <= rd_adr + 8'd1;
            data_1 <= data;
            state <= READ_2;
         end
      READ_2:
         state <= READ_3;
      READ_3:
         begin
            rd_cnt <= rd_cnt + 9'd1;
            rd_adr <= rd_adr + 8'd1;
            data_2 <= data;
            state <= WRITE_INIT;
         end
      WRITE_INIT:
         begin
            wr_out <= 1'b1;
            wr_cnt <= wr_cnt + 9'd1;
            state <= WRITE;
            case (wrd)  // synthesis full_case
               wrd_1:
                  begin
                     wrd <= wrd_2;
                     dout <= data_1 [15:0];
                  end
               wrd_2:
                  begin
                     wrd <= wrd_3;
                     dout <= {data_2 [7:0] , data_1 [23:16]};
                  end
               wrd_3:
                  begin
                     wrd <= wrd_1;
                     dout <= data_2 [23:8];
                  end
            endcase
         end
      WRITE:
         begin
            wr_out <= 1'b0;
            ful_out <= wr_cnt [8] ? 1'b1 : ful_out;
            wr_cnt <= wr_cnt [8] ? 9'd0 : wr_cnt;
            state <= wr_cnt [8] ? BUF_FUL : WRITE_DONE;
         end
      BUF_FUL:
         if (nxt_in)
            begin
               ful_out <= 1'b0;
               state <= WRITE_DONE;
            end
      WRITE_DONE:
         state <= wrd [0] ? READ_INIT : WRITE_INIT;
   endcase
end
```

```
endmodule

module pio_ctrl(
                clk,
                clk_en_2,
                iordy,
                wr_en,
                buf_ful,
                ram_bsy,
                wr_data,
                lba_strt,
                cf_data,
                ncs0,
                ncs1,
                ndmack,
                nreset,
                niord,
                niowr,
                buf_nxt,
                cf_adr,
                mode,
                bsy
                );

   input clk;
   input clk_en_2;
   input iordy;
   input wr_en;
   input buf_ful;
   input ram_bsy;
   input [23:0] wr_data;
   input [27:0] lba_strt;

   inout [15:0] cf_data;

   output ncs0;
   output ncs1;
   output ndmack;
   output nreset;
   output niord;
   output niowr;
   output buf_nxt;
   output [2:0] cf_adr;
   output [2:0] mode;

   output reg bsy;            // synthesis attribute init of bsy is "0"

   reg cf_rset_strt;          // synthesis attribute init of cf_rset_strt is "0"
   reg pio_mode_strt;         // synthesis attribute init of pio_mode_strt is "0"
   reg pio_ctrl_pio_rw_strt;  // synthesis attribute init of pio_ctrl_pio_rw_strt is "0"
   reg [8:0] state;           // synthesis attribute init of state is "000000001"
   reg [27:0] lba;            // synthesis attribute init of lba is
"0000000000000000000000000000"

   wire rw_wr_en;
   wire cf_rset_bsy;
   wire rw_buf_ful;
   wire rw_buf_nxt;
   wire pio_strt;
   wire cf_rset_pio_strt;
   wire pio_rw_pio_strt;
   wire pio_mode_pio_rw_strt;
   wire pio_wr;
   wire cf_rset_pio_wr;
   wire pio_rw_pio_wr;
   wire pio_mode_wr;
   wire pio_ctrl_wr;
   wire pio_bsy;
   wire pio_mode_bsy;
   wire pio_rw_bsy;
   wire [2:0] word;
```

```
wire [2:0] pio_adr;
wire [2:0] cf_rset_pio_adr;
wire [2:0] pio_rw_pio_adr;
wire [7:0] pio_rd_data;
wire [8:0] wrd_num;
wire [8:0] num_sec;
wire [15:0] pio_wr_data;
wire [15:0] rw_wr_data;
wire [27:0] lba_next;

parameter RSET_STRT  = 9'b000000001;
parameter RSET_BUSY  = 9'b000000010;
parameter MODE_STRT  = 9'b000000100;
parameter MODE_BUSY  = 9'b000001000;
parameter IDLE       = 9'b000010000;
parameter WAIT_BUFF  = 9'b000100000;
parameter WRITE_STRT = 9'b001000000;
parameter WRITE_BUSY = 9'b010000000;
parameter WRITE_DONE = 9'b100000000;

assign ncs0 = 1'b0;
assign ncs1 = 1'b1;
assign ndmack = 1'b1;
assign num_sec = 9'd192;
assign pio_ctrl_wr = 1'b1;
assign lba_next = lba + {19'd0 , num_sec};
assign wr = pio_mode_bsy ? pio_mode_wr : pio_ctrl_wr;
assign pio_wr = cf_rset_bsy ? cf_rset_pio_wr : pio_rw_pio_wr;
assign pio_adr = cf_rset_bsy ? cf_rset_pio_adr : pio_rw_pio_adr;
assign pio_strt = cf_rset_bsy ? cf_rset_pio_strt : pio_rw_pio_strt;
assign pio_rw_strt = pio_mode_bsy ? pio_mode_pio_rw_strt : pio_ctrl_pio_rw_strt;

pio pio(
        .clk(clk),
        .clk_en_2(clk_en_2),
        .strt(pio_strt),
        .wr(pio_wr),
        .iordy(iordy),
        .mode(mode),
        .adr_in(pio_adr),
        .wr_data(pio_wr_data),
        .cf_data(cf_data),
        .bsy(pio_bsy),
        .niord(niord),
        .niowr(niowr),
        .adr_out(cf_adr),
        .rd_data(pio_rd_data)
        );

cf_rset cf_rset(
                .clk(clk),
                .clk_en_2(clk_en_2),
                .strt(cf_rset_strt),
                .pio_bsy(pio_bsy),
                .pio_rd_data(pio_rd_data [7:0]),
                .nreset(nreset),
                .bsy(cf_rset_bsy),
                .pio_strt(cf_rset_pio_strt),
                .pio_wr(cf_rset_pio_wr),
                .pio_adr(cf_rset_pio_adr)
                );

pio_mode pio_mode(
                  .clk(clk),
                  .clk_en_2(clk_en_2),
                  .strt(pio_mode_strt),
                  .rw_bsy(pio_rw_bsy),
                  .wrd(word),
                  .wr(pio_mode_wr),
                  .rw_strt(pio_mode_pio_rw_strt),
                  .bsy(pio_mode_bsy),
```

```
                .mode(mode),
                .wrd_num(wrd_num)
                );

pio_buff pio_buff(
                .clk(clk),
                .clk_en_2(clk_en_2),
                .wr_in(wr_en),
                .wr_out(rw_wr_en),
                .din(wr_data),
                .dout(rw_wr_data),
                .ful_in(buf_ful),
                .ful_out(rw_buf_ful),
                .nxt_in(rw_buf_nxt),
                .nxt_out(buf_nxt)
                );

pio_rw pio_rw(
              .clk(clk),
              .clk_en_2(clk_en_2),
              .strt(pio_rw_strt),
              .wr(wr),
              .wr_en(rw_wr_en),
              .pio_bsy(pio_bsy),
              .num_sec(num_sec [7:0]),
              .wrd_num(wrd_num),
              .pio_rd_data(pio_rd_data [7:0]),
              .wr_data(rw_wr_data),
              .lba(lba),
              .buf_ful(rw_buf_ful),
              .buf_nxt(rw_buf_nxt),
              .bsy(pio_rw_bsy),
              .pio_strt(pio_rw_pio_strt),
              .pio_wr(pio_rw_pio_wr),
              .pio_adr(pio_rw_pio_adr),
              .pio_wr_data(pio_wr_data),
              .word(word)
              );

always @(posedge clk)
   if (clk_en_2)
      case (state)   // synthesis full_case
         RSET_STRT:
            begin
               cf_rset_strt <= cf_rset_bsy ? 1'b0 : 1'b1;
               state <= cf_rset_bsy ? RSET_BUSY : state;
            end
         RSET_BUSY:
            if (~cf_rset_bsy)
               begin
                  pio_mode_strt <= 1'b1;
                  state <= MODE_STRT;
               end
         MODE_STRT:
            if (pio_mode_bsy)
               begin
                  pio_mode_strt <= 1'b0;
                  state <= MODE_BUSY;
               end
         MODE_BUSY:
            if (~pio_mode_bsy)
               state <= IDLE;
         IDLE:
            if (ram_bsy)
               begin
                  lba <= lba_strt;
                  bsy <= 1'b1;
                  state <= WAIT_BUFF;
               end
         WAIT_BUFF:
            if (rw_buf_ful)
```

```
                  begin
                     pio_ctrl_pio_rw_strt <= 1'b1;
                     state <= WRITE_STRT;
                  end
            WRITE_STRT:
               if (pio_rw_bsy)
                  begin
                     pio_ctrl_pio_rw_strt <= 1'b0;
                     state <= WRITE_BUSY;
                  end
            WRITE_BUSY:
               if (~pio_rw_bsy)
                  state <= WRITE_DONE;
            WRITE_DONE:
               begin
                  lba <= lba_next;
                  bsy <= ram_bsy ? bsy : 1'b0;
                  state <= ram_bsy ? WAIT_BUFF : IDLE;
               end
         endcase

endmodule

module pio_init(
                clk,
                clk_en_2,
                strt,
                pio_bsy,
                num_sec,
                lba,
                pio_wr,
                pio_strt,
                bsy,
                adr,
                data
                );

   input clk;
   input clk_en_2;
   input strt;
   input pio_bsy;
   input [7:0] num_sec;
   input [27:0] lba;

   output pio_wr;

   output reg pio_strt;    // synthesis attribute init of pio_strt is "0"
   output reg bsy;         // synthesis attribute init of bsy is "0"
   output reg [2:0] adr;   // synthesis attribute init of adr is "010"
   output reg [7:0] data;

   reg [2:0] state;        // synthesis attribute init of state is "001"

   parameter IDLE    = 3'b001;
   parameter STRT    = 3'b010;
   parameter BUSY    = 3'b100;
   parameter sec_cnt = 3'd2;
   parameter sec_num = 3'd3;
   parameter cyl_lo  = 3'd4;
   parameter cyl_hi  = 3'd5;
   parameter drv_hd  = 3'd6;

   assign pio_wr = 1'b1;

   always @(posedge clk)
      if (clk_en_2)
         case (state)   // synthesis full_case
            IDLE:
               if (strt)
                  begin
                     bsy <= 1'b1;
```

```
                  pio_strt <= 1'b1;
                  data <= num_sec;
                  state <= STRT;
               end
            STRT:
               if (pio_bsy)
                  begin
                  pio_strt <= 1'b0;
                  state <= BUSY;
               end
            BUSY:
               if (~pio_bsy)
                  begin
                  pio_strt <= adr == drv_hd ? pio_strt : 1'b1;
                  state <= adr == drv_hd ? IDLE : STRT;
                  case (adr)  // synthesis full_case
                     sec_cnt:
                        begin
                           adr <= sec_num;
                           data <= lba [7:0];
                        end
                     sec_num:
                        begin
                           adr <= cyl_lo;
                           data <= lba [15:8];
                        end
                     cyl_lo:
                        begin
                           adr <= cyl_hi;
                           data <= lba [23:16];
                        end
                     cyl_hi:
                        begin
                           adr <= drv_hd;
                           data <= { 4'd14 , lba [27:24] };
                        end
                     drv_hd:
                        begin
                           bsy <= 1'b0;
                           adr <= sec_cnt;
                        end
                  endcase
               end
         endcase

endmodule

module pio_mode(
                clk,
                clk_en_2,
                strt,
                rw_bsy,
                wrd,
                wr,
                rw_strt,
                bsy,
                mode,
                wrd_num
                );

   input clk;
   input clk_en_2;
   input strt;
   input rw_bsy;
   input [2:0] wrd;

   output wr;

   output reg rw_strt;        // synthesis attribute init of rw_strt is "0"
   output reg bsy;            // synthesis attribute init of bsy is "0"
   output reg [2:0] mode;     // synthesis attribute init of mode is "000"
```

```
output reg [8:0] wrd_num;  // synthesis attribute init of wrd_num is "000110110"

reg [7:0] state;           // synthesis attribute init of state is "00000001"
reg [8:0] count;           // synthesis attribute init of count is "000000000"

wire word;
wire done;

parameter IDLE       = 8'b00000001;
parameter STRT       = 8'b00000010;
parameter BUSY       = 8'b00000100;
parameter DONE       = 8'b00001000;
parameter WORD       = 8'b00010000;
parameter WORD_53    = 8'b00100000;
parameter WORD_64    = 8'b01000000;
parameter WORD_163   = 8'b10000000;
parameter word_53    = 9'd54;
parameter word_64    = 9'd65;
parameter word_163   = 9'd164;

assign wr = 1'b0;
assign done = count [8];
assign word = count == wrd_num;

always @(posedge clk)
   if (clk_en_2)
      case (state)   // synthesis full_case
         IDLE:
            if (strt)
               begin
                  rw_strt <= 1'b1;
                  bsy <= 1'b1;
                  state <= STRT;
               end
         STRT:
            if (rw_bsy)
               begin
                  rw_strt <= 1'b0;
                  count <= count + 9'd1;
                  state <= BUSY;
               end
         BUSY:
            if (~rw_bsy)
               begin
                  rw_strt <= word | done ? rw_strt : 1'b1;
                  state <= word ? WORD : done ? DONE : STRT;
               end
         DONE:
            begin
               bsy <= 1'b0;
               count <= 9'd0;
               state <= IDLE;
            end
         WORD:
            case (wrd_num) // synthesis full_case
               word_53:
                  state <= WORD_53;
               word_64:
                  state <= WORD_64;
               word_163:
                  state <= WORD_163;
            endcase
         WORD_53:
            begin
               rw_strt <= done ? rw_strt : 1'b1;
               wrd_num <= wrd [1] ? word_64 : word_53;
               state <= done ? DONE : STRT;
            end
         WORD_64:
            begin
               rw_strt <= done ? rw_strt : 1'b1;
```

```
                  mode <= wrd [1] ? 3'd4 : wrd [0] ? 3'd3 : mode;
                  wrd_num <= wrd [1] ? word_163 : word_53;
                  state <= done ? DONE : STRT;
               end
            WORD_163:
               begin
                  rw_strt <= done ? rw_strt : 1'b1;
                  wrd_num <= word_53;
                  state <= done ? DONE : STRT;
                  case (wrd)  // synthesis full_case
                     3'd0:
                        mode <= mode;
                     3'd1:
                        mode <= 3'd5;
                     3'd2:
                        mode <= 3'd6;
                     3'd3:
                        mode <= mode;
                     3'd4:
                        mode <= mode;
                     3'd5:
                        mode <= mode;
                     3'd6:
                        mode <= mode;
                     3'd7:
                        mode <= mode;
                  endcase
               end
         endcase

endmodule

module pio_rw(
              clk,
              clk_en_2,
              strt,
              wr,
              wr_en,
              pio_bsy,
              buf_ful,
              buf_nxt,
              num_sec,
              wrd_num,
              pio_rd_data,
              wr_data,
              lba,
              bsy,
              pio_strt,
              pio_wr,
              pio_adr,
              pio_wr_data,
              word
              );

   input clk;
   input clk_en_2;
   input strt;
   input wr;
   input wr_en;
   input pio_bsy;
   input buf_ful;
   input [7:0] num_sec;
   input [7:0] pio_rd_data;
   input [8:0] wrd_num;
   input [15:0] wr_data;
   input [27:0] lba;

   output pio_strt;
   output pio_wr;
   output [2:0] pio_adr;
   output [15:0] pio_wr_data;
```

```
   output reg buf_nxt;              // synthesis attribute init of buf_nxt is "0"
   output reg bsy;                  // synthesis attribute init of bsy is "0"
   output reg [2:0] word;           // synthesis attribute init of word is "000"

   reg cmd;                         // synthesis attribute init of cmd is "1"
   reg rd_en;                       // synthesis attribute init of rd_en is "0"
   reg buf_sel;                     // synthesis attribute init of buf_sel is "0"
   reg pio_init_strt;               // synthesis attribute init of pio_init_strt is "0"
   reg pio_rw_pio_strt;             // synthesis attribute init of pio_rw_pio_strt is "0"
   reg pio_rw_pio_wr;               // synthesis attribute init of pio_rw_pio_wr is "0"
   reg [2:0] pio_rw_pio_adr;        // synthesis attribute init of pio_rw_pio_adr is
"111"
   reg [7:0] sec_cnt;               // synthesis attribute init of sec_cnt is "00000001"
   reg [7:0] rd_adr;                // synthesis attribute init of rd_adr is "00000000"
   reg [7:0] wr_adr;                // synthesis attribute init of wr_adr is "00000000"
   reg [8:0] wrd_cnt;               // synthesis attribute init of wrd_cnt is "000000000"
   reg [14:0] state;                // synthesis attribute init of state is
"000000000000001"
   reg [15:0] rd_data_0;            // synthesis attribute init of rd_data_0 is
"0000000000000000"
   reg [15:0] rd_data_1;            // synthesis attribute init of rd_data_1 is
"0000000000000000"
   reg [15:0] buf_0 [0:255];        // synthesis attribute ram_style of buf_0 is block
   reg [15:0] buf_1 [0:255];        // synthesis attribute ram_style of buf_1 is block

   wire rrdy;
   wire wrdy;
   wire wrd_done;
   wire sec_done;
   wire sec_inc;
   wire pio_init_pio_wr;
   wire pio_init_pio_strt;
   wire pio_init_bsy;
   wire nbsy;
   wire drdy;
   wire [2:0] pio_init_pio_adr;
   wire [7:0] pio_init_pio_wr_data;
   wire [15:0] pio_rw_pio_wr_data;

   parameter IDLE          = 15'b000000000000001;
   parameter INIT_STRT     = 15'b000000000000010;
   parameter INIT_BUSY     = 15'b000000000000100;
   parameter STAT_STRT_1   = 15'b000000000001000;
   parameter STAT_BUSY_1   = 15'b000000000010000;
   parameter CMND_STRT     = 15'b000000000100000;
   parameter CMND_BUSY     = 15'b000000001000000;
   parameter STAT_STRT_2   = 15'b000000010000000;
   parameter STAT_BUSY_2   = 15'b000000100000000;
   parameter DATA_INIT     = 15'b000001000000000;
   parameter DATA_STRT     = 15'b000010000000000;
   parameter DATA_BUSY     = 15'b000100000000000;
   parameter DATA_DONE     = 15'b001000000000000;
   parameter STAT_STRT_3   = 15'b010000000000000;
   parameter STAT_BUSY_3   = 15'b100000000000000;
   parameter data          = 3'd0;
   parameter stat          = 3'd7;
   parameter ident_dev     = 16'h00ec;
   parameter read_sec      = 16'h0020;
   parameter write_sec     = 16'h0030;

   assign rrdy = drdy & ~ wr;
   assign wrd_done = wrd_cnt [8];
   assign sec_inc = wr & wrd_done;
   assign wrdy = buf_ful & drdy & wr;
   assign nbsy = pio_rd_data == 8'h50;
   assign drdy = pio_rd_data == 8'h58;
   assign sec_done = wr ? sec_cnt == num_sec : sec_cnt == 8'd1;
   assign pio_wr = pio_init_bsy ? pio_init_pio_wr : pio_rw_pio_wr;
   assign pio_adr = pio_init_bsy ? pio_init_pio_adr : pio_rw_pio_adr;
   assign pio_strt = pio_init_bsy ? pio_init_pio_strt : pio_rw_pio_strt;
```

```
   assign pio_wr_data = pio_init_bsy ? { 7'd0 , pio_init_pio_wr_data } :
pio_rw_pio_wr_data;
   assign pio_rw_pio_wr_data = cmd ? (wr ? write_sec : ident_dev) : (buf_sel ? rd_data_0
: rd_data_1);

   pio_init pio_init(
                    .clk(clk),
                    .clk_en_2(clk_en_2),
                    .strt(pio_init_strt),
                    .pio_bsy(pio_bsy),
                    .num_sec(num_sec),
                    .lba(lba),
                    .pio_wr(pio_init_pio_wr),
                    .pio_strt(pio_init_pio_strt),
                    .bsy(pio_init_bsy),
                    .adr(pio_init_pio_adr),
                    .data(pio_init_pio_wr_data)
                    );

   always @(posedge clk)
      if (clk_en_2)
         begin
            rd_adr <= rd_en ? rd_adr + 8'd1 : rd_adr;
            rd_data_0 <= rd_en ? buf_0 [rd_adr] : rd_data_0;
            rd_data_1 <= rd_en ? buf_1 [rd_adr] : rd_data_1;
            if (wr_en)
               begin
                  wr_adr <= wr_adr + 8'd1;
                  buf_0 [wr_adr] <= buf_sel ? buf_0 [wr_adr] : wr_data;
                  buf_1 [wr_adr] <= buf_sel ? wr_data : buf_1 [wr_adr];
               end
            case (state)   // synthesis full_case
               IDLE:
                  if (strt)
                     begin
                        pio_init_strt <= 1'b1;
                        bsy <= 1'b1;
                        cmd <= 1'b1;
                        sec_cnt <= 8'd1;
                        state <= INIT_STRT;
                     end
               INIT_STRT:
                  if (pio_init_bsy)
                     begin
                        pio_init_strt <= 1'b0;
                        state <= INIT_BUSY;
                     end
               INIT_BUSY:
                  if (~pio_init_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b1;
                        state <= STAT_STRT_1;
                     end
               STAT_STRT_1:
                  if (pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b0;
                        state <= STAT_BUSY_1;
                     end
               STAT_BUSY_1:
                  if (~pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b1;
                        pio_rw_pio_wr <= nbsy ? 1'b1 : pio_rw_pio_wr;
                        state <= nbsy ? CMND_STRT : STAT_STRT_1;
                     end
               CMND_STRT:
                  if (pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b0;
                        state <= CMND_BUSY;
```

```
                  end
               CMND_BUSY:
                  if (~pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b1;
                        pio_rw_pio_wr <= 1'b0;
                        cmd <= 1'b0;
                        state <= STAT_STRT_2;
                     end
               STAT_STRT_2:
                  if (pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b0;
                        rd_adr <= 8'd0;
                        state <= STAT_BUSY_2;
                     end
               STAT_BUSY_2:
                  if (~pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b1;
                        pio_rw_pio_wr <= wrdy ? 1'b1 : pio_rw_pio_wr;
                        pio_rw_pio_adr <= rrdy | wrdy ? data : pio_rw_pio_adr;
                        buf_sel <= wrdy ? ~ buf_sel : buf_sel;
                        buf_nxt <= wrdy ? 1'b1 : buf_nxt;
                        rd_en <= rrdy | wrdy ? 1'b1 : rd_en;
                        state <= rrdy | wrdy ? DATA_INIT : STAT_STRT_2;
                     end
               DATA_INIT:
                  begin
                     rd_en <= 1'b0;
                     buf_nxt <= 1'b0;
                     wrd_cnt <= 9'd0;
                     state <= DATA_STRT;
                  end
               DATA_STRT:
                  if (pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b0;
                        wrd_cnt <= wrd_cnt + 9'd1;
                        state <= DATA_BUSY;
                     end
               DATA_BUSY:
                  if (~pio_bsy)
                     begin
                        rd_en <= 1'b1;
                        state <= DATA_DONE;
                     end
               DATA_DONE:
                  begin
                     rd_en <= 1'b0;
                     pio_rw_pio_strt <= 1'b1;
                     pio_rw_pio_wr <= wrd_done ? 1'b0 : pio_rw_pio_wr;
                     pio_rw_pio_adr <= wrd_done ? stat : pio_rw_pio_adr;
                     sec_cnt <= sec_inc ? sec_cnt + 8'd1 : sec_cnt;
                     state <= wrd_done ? sec_done ? STAT_STRT_3 : STAT_STRT_2 :
DATA_STRT;
                     if (wrd_cnt == wrd_num)
                        word <= wr ? word : pio_rd_data [2:0];
                  end
               STAT_STRT_3:
                  if (pio_bsy)
                     begin
                        pio_rw_pio_strt <= 1'b0;
                        state <= STAT_BUSY_3;
                     end
               STAT_BUSY_3:
                  if (~pio_bsy)
                     begin
                        pio_rw_pio_strt <= nbsy ? pio_rw_pio_strt : 1'b1;
                        bsy <= nbsy ? 1'b0 : bsy;
                        state <= nbsy ? IDLE : STAT_STRT_3;
```

```
                    end
            endcase
         end

endmodule

module pulse(
             clk,
             clk_en_2,
             en,
             invrt,
             wdth,
             trgr,
             puls
             );

   input clk;
   input clk_en_2;
   input en;
   input invrt;
   input [7:0] wdth;

   output reg trgr;  // synthesis attribute init of trgr is "0"
   output reg puls;  // synthesis attribute init of puls is "0"

   reg pulls;        // synthesis attribute init of pulls is "0"
   reg [7:0] count;  // synthesis attribute init of count is "11111111"

   always @(posedge clk)
      if (clk_en_2)
         begin
            trgr <= count ? trgr : ~ trgr;
            pulls <= invrt ? ~ trgr : trgr;
            puls <= en ? pulls : puls;
            count <= count ? count - 8'd1 : (wdth < 8'd5) ? 8'd5 : wdth;
         end

endmodule

module ram(
           clk_in,
           clk_en_2,
           clk_out,
           cke,
           ncs,
           nras,
           ncas,
           nwe,
           dqm,
           badr,
           adr_out,
           dq,
           wr,
           strt,
           bsy,
           adr_in,
           rd_en,
           wr_en,
           rd_data,
           wr_data
           );

   input clk_in;
   input clk_en_2;
   input wr;
   input strt;
   input [23:0] wr_data;
   input [25:0] adr_in;

   inout [23:0] dq;
```

```
    output dqm;
    output cke;
    output ncs;
    output nras;
    output ncas;
    output nwe;
    output [12:0] adr_out;

    output reg clk_out;
    output reg rd_en;          // synthesis attribute init of rd_en is "0"
    output reg wr_en;          // synthesis attribute init of wr_en is "0"
    output reg bsy;            // synthesis attribute init of bsy is "1"
    output reg [1:0] badr;     // synthesis attribute init of badr is "00"
    output reg [23:0] rd_data; // synthesis attribute init of rd_data is
"000000000000000000000000"

    reg ram_nras;              // synthesis attribute init of ram_nras is "1"
    reg ram_ncas;              // synthesis attribute init of ram_ncas is "1"
    reg ram_nwe;               // synthesis attribute init of ram_nwe is "1"
    reg [7:0] count;           // synthesis attribute init of count is "00000000"
    reg [12:0] ram_adr;        // synthesis attribute init of ram_adr is "0000000000000"
    reg [13:0] state;          // synthesis attribute init of state is "00000000000001"

    wire done;
    wire init_bsy;
    wire init_cke;
    wire ram_cke;
    wire init_ncs;
    wire ram_ncs;
    wire init_nras;
    wire init_ncas;
    wire init_nwe;
    wire [12:0] init_adr;

    parameter INIT_BUSY  = 14'b00000000000001;
    parameter REFRESH    = 14'b00000000000010;
    parameter NOP_1      = 14'b00000000000100;
    parameter NOP_2      = 14'b00000000001000;
    parameter NOP_3      = 14'b00000000010000;
    parameter NOP_4      = 14'b00000000100000;
    parameter ACTIVE     = 14'b00000001000000;
    parameter NOP_5      = 14'b00000010000000;
    parameter WRITE      = 14'b00000100000000;
    parameter READ       = 14'b00001000000000;
    parameter BURST_TERM = 14'b00010000000000;
    parameter PRECHARGE  = 14'b00100000000000;
    parameter NOP_6      = 14'b01000000000000;
    parameter NOP_7      = 14'b10000000000000;

    assign dqm = 1'b0;
    assign ram_cke = 1'b1;
    assign ram_ncs = 1'b0;
    assign done = count == 8'd255;
    assign cke = init_bsy ? init_cke : ram_cke;
    assign ncs = init_bsy ? init_ncs : ram_ncs;
    assign nwe = init_bsy ? init_nwe : ram_nwe;
    assign nras = init_bsy ? init_nras : ram_nras;
    assign ncas = init_bsy ? init_ncas : ram_ncas;
    assign adr_out = init_bsy ? init_adr : ram_adr;
    assign dq = state == WRITE ? wr_data : 24'hzzzzzz;

    ram_init ram_init(
                      .clk(clk_in),
                      .clk_en_2(clk_en_2),
                      .bsy(init_bsy),
                      .cke(init_cke),
                      .ncs(init_ncs),
                      .nras(init_nras),
                      .ncas(init_ncas),
                      .nwe(init_nwe),
                      .adr(init_adr)
```

```
                );

always @(posedge clk_in)
   begin
      clk_out <= clk_en_2;
      if (clk_en_2)
         begin
            rd_data <= dq;
            case (state)   // synthesis full_case
               INIT_BUSY:
                  if (~init_bsy)
                     begin
                        ram_nras <= 1'b0;
                        ram_ncas <= 1'b0;
                        bsy <= 1'b0;
                        state <= REFRESH;
                     end
               REFRESH:
                  begin
                     ram_nras <= 1'b1;
                     ram_ncas <= 1'b1;
                     state <= NOP_1;
                  end
               NOP_1:
                  state <= NOP_2;
               NOP_2:
                  state <= NOP_3;
               NOP_3:
                  state <= NOP_4;
               NOP_4:
                  begin
                     ram_nras <= 1'b0;
                     ram_ncas <= strt ? ram_ncas : 1'b0;
                     bsy <= strt ? 1'b1 : bsy;
                     ram_adr <= adr_in [23:11];
                     badr <= adr_in [25:24];
                     state <= strt ? ACTIVE : REFRESH;
                  end
               ACTIVE:
                  begin
                     ram_nras <= 1'b1;
                     state <= NOP_5;
                  end
               NOP_5:
                  begin
                     rd_en <= wr ? 1'b1 : rd_en;
                     ram_ncas <= 1'b0;
                     ram_nwe <= wr ? 1'b0 : ram_nwe;
                     ram_adr [9:0] <= adr_in [9:0];
                     ram_adr [10] <= 1'b0;
                     ram_adr [11] <= adr_in [10];
                     state <= wr ? WRITE : READ;
                  end
               READ:
                  begin
                     wr_en <= & count [1:0] ? 1'b1 : wr_en;
                     ram_ncas <= 1'b1;
                     ram_nwe <= done ? 1'b0 : ram_nwe;
                     count <= count + 8'd1;
                     state <= done ? BURST_TERM : state;
                  end
               WRITE:
                  begin
                     rd_en <= done ? 1'b0 : rd_en;
                     ram_ncas <= 1'b1;
                     ram_nwe <= done ? 1'b0 : 1'b1;
                     count <= count + 8'd1;
                     state <= done ? BURST_TERM : state;
                  end
               BURST_TERM:
                  begin
```

```
                        ram_nras <= 1'b0;
                        state <= PRECHARGE;
                     end
                  PRECHARGE:
                     begin
                        ram_nras <= 1'b1;
                        ram_nwe <= 1'b1;
                        state <= NOP_6;
                     end
                  NOP_6:
                     state <= NOP_7;
                  NOP_7:
                     begin
                        wr_en <= 1'b0;
                        ram_nras <= 1'b0;
                        ram_ncas <= 1'b0;
                        bsy <= 1'b0;
                        state <= REFRESH;
                     end
               endcase
            end
      end

endmodule

module ram_ctrl(
                clk_in,
                clk_en_2,
                strt,
                wr_en,
                buf_nxt,
                blk_num,
                wr_data,
                dq,
                clk_out,
                dqm,
                cke,
                ncs,
                nras,
                ncas,
                nwe,
                bsy,
                pio_wr_en,
                buf_ful,
                badr,
                adr_out,
                rd_data
                );

   input clk_in;
   input clk_en_2;
   input strt;
   input wr_en;
   input buf_nxt;
   input [10:0] blk_num;
   input [23:0] wr_data;

   inout [23:0] dq;

   output clk_out;
   output dqm;
   output cke;
   output ncs;
   output nras;
   output ncas;
   output nwe;
   output pio_wr_en;
   output [1:0] badr;
   output [12:0] adr_out;
   output [23:0] rd_data;
```

```
   output reg bsy;              // synthesis attribute init of bsy is "0"
   output reg buf_ful;          // synthesis attribute init of buf_ful is "0"

   reg ram_ctrl_rd;             // synthesis attribute init of ram_ctrl_rd is "0"
   reg wr;                      // synthesis attribute init of wr is "0"
   reg ram_strt;                // synthesis attribute init of ram_strt is "0"
   reg [7:0] ram_cnt;           // synthesis attribute init of ram_cnt is "00000000"
   reg [8:0] fifo_rd_adr;       // synthesis attribute init of fifo_rd_adr is
"000000000"
   reg [8:0] fifo_wr_adr;       // synthesis attribute init of fifo_wr_adr is
"000000000"
   reg [9:0] fifo_cnt;          // synthesis attribute init of fifo_cnt is "0000000000"
   reg [10:0] num_blk;          // synthesis attribute init of num_blk is "00000000000"
   reg [11:0] blk_cnt;          // synthesis attribute init of blk_cnt is "000000000000"
   reg [15:0] state;            // synthesis attribute init of state is
"0000000000000001"
   reg [17:0] ram_adr;          // synthesis attribute init of ram_adr is
"000000000000000000"
   reg [23:0] ram_wr_data;      // synthesis attribute init of ram_wr_data is
"000000000000000000000000"
   reg [23:0] fifo [0:511];     // synthesis attribute ram_style of fifo is block

   wire ram_bsy;
   wire fifo_rd;
   wire ram_rd;
   wire fifo_rdy;
   wire strt;
   wire ram_done;
   wire blk_done;

   parameter INIT       = 16'b0000000000000001;
   parameter IDLE       = 16'b0000000000000010;
   parameter STRT       = 16'b0000000000000100;
   parameter READ_FIFO  = 16'b0000000000001000;
   parameter WAIT_FIFO  = 16'b0000000000010000;
   parameter WRITE_INIT = 16'b0000000000100000;
   parameter WRITE_STRT = 16'b0000000001000000;
   parameter WRITE_BUSY = 16'b0000000010000000;
   parameter WRITE_DONE = 16'b0000000100000000;
   parameter READ_REDY  = 16'b0000001000000000;
   parameter WAIT_COMP  = 16'b0000010000000000;
   parameter READ_INIT  = 16'b0000100000000000;
   parameter READ_STRT  = 16'b0001000000000000;
   parameter READ_BUSY  = 16'b0010000000000000;
   parameter READ_DONE  = 16'b0100000000000000;
   parameter NEXT_BUFF  = 16'b1000000000000000;

   ram ram(
           .clk_in(clk_in),
           .clk_en_2(clk_en_2),
           .clk_out(clk_out),
           .cke(cke),
           .ncs(ncs),
           .nras(nras),
           .ncas(ncas),
           .nwe(nwe),
           .dqm(dqm),
           .badr(badr),
           .adr_out(adr_out),
           .dq(dq),
           .wr(wr),
           .strt(ram_strt),
           .bsy(ram_bsy),
           .adr_in({ram_adr , 8'd0}),
           .rd_en(ram_rd),
           .wr_en(pio_wr_en),
           .rd_data(rd_data),
           .wr_data(ram_wr_data)
           );

   assign fifo_rdy = fifo_cnt > 10'd255;
```

```
   assign fifo_rd = ram_rd | ram_ctrl_rd;
   assign ram_done = ram_cnt [7];
   assign blk_done = blk_cnt == {1'd0 , num_blk} + 12'd1;

   always @(posedge clk_in)
      if (clk_en_2)
         begin
            fifo_cnt <= wr_en & ~ fifo_rd ? fifo_cnt + 10'd1 : fifo_rd & ~ wr_en ?
fifo_cnt - 10'd1 : fifo_cnt;
            ram_wr_data <= fifo_rd ? fifo [fifo_rd_adr] : ram_wr_data;
            fifo_rd_adr <= fifo_rd ? fifo_rd_adr + 9'd1 : fifo_rd_adr;
            fifo_wr_adr <= wr_en ? fifo_wr_adr + 9'd1 : fifo_wr_adr;
            fifo [fifo_wr_adr] <= wr_en ? wr_data : fifo [fifo_wr_adr];
            case (state)   // synthesis full_case
               INIT:
                  if (~ram_bsy)
                     state <= IDLE;
               IDLE:
                  begin
                     ram_cnt <= 8'd0;
                     fifo_rd_adr <= 9'd0;
                     fifo_wr_adr <= 9'd0;
                     fifo_cnt <= 10'd0;
                     blk_cnt <= 12'd0;
                     ram_adr <= 18'd0;
                     num_blk <= blk_num;
                     bsy <= strt ? 1'b1 : bsy;
                     state <= strt ? STRT : state;
                  end
               STRT:
                  begin
                     ram_ctrl_rd <= fifo_cnt ? 1'b1 : ram_ctrl_rd;
                     state <= fifo_cnt ? READ_FIFO : state;
                  end
               READ_FIFO:
                  begin
                     ram_ctrl_rd <= 1'b0;
                     state <= WAIT_FIFO;
                  end
               WAIT_FIFO:
                  state <= fifo_rdy ? WRITE_INIT : state;
               WRITE_INIT:
                  begin
                     wr <= 1'b1;
                     ram_strt <= blk_done ? ram_strt : 1'b1;
                     state <= blk_done ? READ_REDY : WRITE_STRT;
                  end
               WRITE_STRT:
                  if (ram_bsy)
                     begin
                        ram_strt <= 1'b0;
                        state <= WRITE_BUSY;
                     end
               WRITE_BUSY:
                  if (~ram_bsy)
                     begin
                        ram_cnt <= ram_cnt + 8'd1;
                        ram_adr <= ram_adr + 18'd1;
                        state <= WRITE_DONE;
                     end
               WRITE_DONE:
                  begin
                     ram_cnt <= ram_done ? 8'd0 : ram_cnt;
                     blk_cnt <= ram_done ? blk_cnt + 12'd1 : blk_cnt;
                     state <= WAIT_FIFO;
                  end
               READ_REDY:
                  begin
                     blk_cnt <= 12'd0;
                     ram_adr <= 18'd0;
                     state <= WAIT_COMP;
```

```
                end
             WAIT_COMP:
                state <= READ_INIT;
             READ_INIT:
                begin
                   wr <= 1'b0;
                   ram_strt <= blk_done ? ram_strt : 1'b1;
                   bsy <= blk_done ? 1'b0 : bsy;
                   state <= blk_done ? IDLE : READ_STRT;
                end
             READ_STRT:
                if (ram_bsy)
                   begin
                      ram_strt <= 1'b0;
                      state <= READ_BUSY;
                   end
             READ_BUSY:
                if (~ram_bsy)
                   begin
                      buf_ful <= 1'b1;
                      ram_cnt <= ram_cnt + 8'd1;
                      ram_adr <= ram_adr + 18'd1;
                      state <= READ_DONE;
                   end
             READ_DONE:
                if (buf_nxt)
                   begin
                      buf_ful <= 1'b0;
                      state <= NEXT_BUFF;
                   end
             NEXT_BUFF:
                begin
                   ram_cnt <= ram_done ? 8'd0 : ram_cnt;
                   blk_cnt <= ram_done ? blk_cnt + 12'd1 : blk_cnt;
                   state <= WAIT_COMP;
                end
          endcase
       end

endmodule

module ram_init(
                clk,
                clk_en_2,
                bsy,
                cke,
                ncs,
                nras,
                ncas,
                nwe,
                adr
                );

   input clk;
   input clk_en_2;

   output [12:0] adr;

   output reg bsy;         // synthesis attribute init of bsy is "1"
   output reg cke;         // synthesis attribute init of cke is "0"
   output reg ncs;         // synthesis attribute init of ncs is "1"
   output reg nras;        // synthesis attribute init of nras is "1"
   output reg ncas;        // synthesis attribute init of ncas is "1"
   output reg nwe;         // synthesis attribute init of nwe is "1"

   reg a10;                // synthesis attribute init of a10 is "1"
   reg [8:0] state;        // synthesis attribute init of state is "000000001"
   reg [13:0] count;       // synthesis attribute init of count is "00000000000000"

   parameter PWR_UP        = 9'b000000001;
   parameter NOP_1         = 9'b000000010;
```

```
parameter PRECHARGE     = 9'b000000100;
parameter NOP_2         = 9'b000001000;
parameter REFRESH_1     = 9'b000010000;
parameter NOP_3         = 9'b000100000;
parameter REFRESH_2     = 9'b001000000;
parameter NOP_4         = 9'b010000000;
parameter LOAD_MODE_REG = 9'b100000000;
parameter bl            = 3'b111;
parameter bt            = 1'b0;
parameter cl            = 3'b011;
parameter om            = 2'b00;
parameter wb            = 1'b0;

assign adr = { 2'b00 , a10 , wb , om , cl , bt , bl };

always @(posedge clk)
   if (clk_en_2)
      begin
         count <= count + 14'd1;
         case (state)   // synthesis full_case
            PWR_UP:
               begin
                  bsy <= 1'b1;
                  cke <= count [13] ? 1'b1 : 1'b0;
                  ncs <= count [13] ? 1'b0 : 1'b1;
                  a10 <= 1'b1;
                  state <= count [13] ? NOP_1 : state;
               end
            NOP_1:
               begin
                  nras <= 1'b0;
                  nwe <= 1'b0;
                  state <= PRECHARGE;
               end
            PRECHARGE:
               begin
                  nras <= 1'b1;
                  nwe <= 1'b1;
                  state <= NOP_2;
               end
            NOP_2:
               begin
                  nras <= 1'b0;
                  ncas <= 1'b0;
                  state <= REFRESH_1;
               end
            REFRESH_1:
               begin
                  nras <= 1'b1;
                  ncas <= 1'b1;
                  count <= 14'd0;
                  state <= NOP_3;
               end
            NOP_3:
               if (count [2])
                  begin
                     nras <= 1'b0;
                     ncas <= 1'b0;
                     state <= REFRESH_2;
                  end
            REFRESH_2:
               begin
                  nras <= 1'b1;
                  ncas <= 1'b1;
                  count <= 14'd0;
                  state <= NOP_4;
               end
            NOP_4:
               if (count [2])
                  begin
                     nras <= 1'b0;
```

```
                        ncas <= 1'b0;
                        nwe <= 1'b0;
                        a10 <= 1'b0;
                        count <= 14'd0;
                        state <= LOAD_MODE_REG;
                     end
                  LOAD_MODE_REG:
                     begin
                        nras <= 1'b1;
                        ncas <= 1'b1;
                        nwe <= 1'b1;
                        bsy <= count [0] ? 1'b0 : bsy;
                        count <= count [0] ? count : count + 14'd1;
                     end
               endcase
            end

endmodule

module reg_bank(
                clk,
                clk_en_2,
                reg_strt,
                wr_en,
                addr,
                wr_data,
                rd_data,
                invrt,
                mode,
                acq_strt,
                blk,
                lba,
                acq_bsy,
                pio_mode,
                step,
                wdth,
                sampls,
                adc_dlay,
                buf_wrd_num,
                attn,
                rom_ctrl_bsy,
                load,
                reg_bsy,
                acq_on,
                acq_off,
                armd,
                disarm,
                tmpgd,
                dval,
                fstrt,
                fbsy,
                fval,
                tmpen
                );

   input clk;
   input clk_en_2;
   input reg_strt;
   input wr_en;
   input acq_bsy;
   input rom_ctrl_bsy;
   input disarm;
   input tmpgd;
   input fbsy;
   input [2:0] pio_mode;
   input [4:0] addr;
   input [7:0] wr_data;
   input [19:0] fval;

   output load;
   output fstrt;
```

```
output acq_strt;
output invrt;
output armd;
output tmpen;
output [2:0] mode;
output [1:0] step;
output [5:0] attn;
output [7:0] wdth;
output [7:0] sampls;
output [7:0] adc_dlay;
output [7:0] acq_on;
output [7:0] acq_off;
output [9:0] dval;
output [10:0] blk;
output [11:0] buf_wrd_num;
output [27:0] lba;

output reg reg_bsy;        // synthesis attribute init of reg_bsy is "0"
output reg [7:0] rd_data;  // synthesis attribute init of rd_data is "00000000"

reg tmp_en;                // synthesis attribute init of tmp_en is "0"
reg fstart;                // synthesis attribute init of fstart is "0"
reg [7:0] dval_lo;         // synthesis attribute init of dval_lo is "00000000"
reg [1:0] dval_hi;         // synthesis attribute init of dval_hi is "00"
reg tmpgd1;                // tmpgd synchronization register
reg tmpgd2;                // tmpgd synchronization register
reg armed;                 // synthesis attribute init of armed is "0"
reg [7:0] aon;             // synthesis attribute init of aon is "00000000"
reg [7:0] aoff;            // synthesis attribute init of aoff is "00000000"
reg [3:0] state;           // synthesis attribute init of state is "0001"
reg lod;                   // synthesis attribute init of lod is "0"
reg [5:0] atn;             // synthesis attribute init of atn is "000000"
reg strt;                  // synthesis attribute init of strt is "0"
reg invert;                // synthesis attribute init of invert is "0"
reg [2:0] moad;            // synthesis attribute init of moad is "000"
reg [1:0] dstep;           // synthesis attribute init of dstep is "00"
reg [7:0] width;           // synthesis attribute init of width is "00000000"
reg [7:0] samples;         // synthesis attribute init of samples is "00000000"
reg [7:0] adc_delay;       // synthesis attribute init of adc_delay is "00000000"
reg [7:0] lba_0;           // synthesis attribute init of lba_0 is "00000000"
reg [7:0] lba_1;           // synthesis attribute init of lba_1 is "00000000"
reg [7:0] lba_2;           // synthesis attribute init of lba_2 is "00000000"
reg [3:0] lba_3;           // synthesis attribute init of lba_3 is "0000"
reg [7:0] blk_0;           // synthesis attribute init of blk_0 is "00000000"
reg [2:0] blk_1;           // synthesis attribute init of blk_1 is "000"
reg [7:0] buf_wrd_num_0;   // synthesis attribute init of buf_wrd_num_0 is "00000000"
reg [3:0] buf_wrd_num_1;   // synthesis attribute init of buf_wrd_num_1 is "0000"

parameter IDLE    = 4'b0001;
parameter READ    = 4'b0010;
parameter WRITE   = 4'b0100;
parameter DONE    = 4'b1000;

assign tmpen = tmp_en ? 1'bz : 1'b0;
assign armd = armed;
assign acq_on = aon;
assign acq_off = aoff;
assign load = lod;
assign attn = atn;
assign step = dstep;
assign wdth = width;
assign invrt = invert;
assign mode = moad;
assign fstrt = fstart;
assign acq_strt = strt;
assign sampls = samples;
assign adc_dlay = adc_delay;
assign blk = {blk_1 , blk_0};
assign lba = {lba_3 , lba_2 , lba_1 , lba_0};
assign buf_wrd_num = {buf_wrd_num_1 , buf_wrd_num_0};
assign dval = {dval_hi , dval_lo};
```

```
always @(posedge clk)
   if (clk_en_2)
      begin
         tmpgd1 <= tmpgd;  // synchronize tmpgd
         tmpgd2 <= tmpgd1; // synchronize tmpgd
         lod <= rom_ctrl_bsy ? 1'b0 : lod;
         strt <= acq_bsy ? 1'b0 : strt;
         fstart <= fbsy ? 1'b0 : fstart;
         armed <= disarm ? 1'b0 : armed;
         case (state)   // synthesis full_case
            IDLE:
               if (reg_strt)
                  begin
                     reg_bsy <= 1'b1;
                     state <= wr_en ? WRITE : READ;
                  end
            READ:
               begin
                  reg_bsy <= 1'b0;
                  state <= DONE;
                  case (addr) // synthesis full_case
                     5'd0:
                        rd_data <= {7'd0 , acq_bsy};
                     5'd1:
                        rd_data <= {7'd0 , strt};
                     5'd2:
                        rd_data <= {5'd0 , moad};
                     5'd3:
                        rd_data <= {5'd0 , pio_mode};
                     5'd4:
                        rd_data <= lba_0;
                     5'd5:
                        rd_data <= lba_1;
                     5'd6:
                        rd_data <= lba_2;
                     5'd7:
                        rd_data <= {4'd0 , lba_3};
                     5'd8:
                        rd_data <= blk_0;
                     5'd9:
                        rd_data <= {5'd0 , blk_1};
                     5'd10:
                        rd_data <= {6'd0 , dstep};
                     5'd11:
                        rd_data <= width;
                     5'd12:
                        rd_data <= samples;
                     5'd13:
                        rd_data <= adc_delay;
                     5'd14:
                        rd_data <= buf_wrd_num_0;
                     5'd15:
                        rd_data <= {4'd0 , buf_wrd_num_1};
                     5'd16:
                        rd_data <= {7'd0 , invert};
                     5'd17:
                        rd_data <= {2'd0 , atn};
                     5'd18:
                        rd_data <= {7'd0 , lod};
                     5'd19:
                        rd_data <= {7'd0 , rom_ctrl_bsy};
                     5'd20:
                        rd_data <= aoff;
                     5'd21:
                        rd_data <= aon;
                     5'd22:
                        rd_data <= {7'd0 , armed};
                     5'd23:
                        rd_data <= {7'd0 , tmpgd2};
                     5'd24:
```

```
            rd_data <= dval_lo;
         5'd25:
            rd_data <= {6'd0 , dval_hi};
         5'd26:
            rd_data <= {7'd0 , fstart};
         5'd27:
            rd_data <= {7'd0 , fbsy};
         5'd28:
            rd_data <= fval [7:0];
         5'd29:
            rd_data <= fval [15:8];
         5'd30:
            rd_data <= {4'd0 , fval [19:16]};
         5'd31:
            rd_data <= {7'd0 , tmp_en};
      endcase
   end
WRITE:
   begin
      reg_bsy <= 1'b0;
      state <= DONE;
      case (addr) // synthesis full_case
         5'd1:
            strt <= wr_data [0];
         5'd2:
            moad <= wr_data [2:0];
         5'd4:
            lba_0 <= wr_data;
         5'd5:
            lba_1 <= wr_data;
         5'd6:
            lba_2 <= wr_data;
         5'd7:
            lba_3 <= wr_data [3:0];
         5'd8:
            blk_0 <= wr_data;
         5'd9:
            blk_1 <= wr_data [2:0];
         5'd10:
            dstep <= wr_data [1:0];
         5'd11:
            width <= wr_data;
         5'd12:
            samples <= wr_data;
         5'd13:
            adc_delay <= wr_data;
         5'd14:
            buf_wrd_num_0 <= wr_data;
         5'd15:
            buf_wrd_num_1 <= wr_data [3:0];
         5'd16:
            invert <= wr_data [0];
         5'd17:
            atn <= wr_data [5:0];
         5'd18:
            lod <= wr_data [0];
         5'd20:
            aoff <= wr_data;
         5'd21:
            aon <= wr_data;
         5'd22:
            armed <= wr_data [0];
         5'd24:
            dval_lo <= wr_data;
         5'd25:
            dval_hi <= wr_data [1:0];
         5'd26:
            fstart <= wr_data [0];
         5'd31:
            tmp_en <= wr_data [0];
      endcase
```

```
                end
             DONE:
                state <= IDLE;
          endcase
       end

endmodule

module reg_mux(
             clk,
             clk_en_2,
             rom_ctrl_bsy,
             uart_ctrl_reg_strt,
             rom_ctrl_reg_strt,
             uart_ctrl_reg_wr,
             rom_ctrl_reg_wr,
             uart_ctrl_reg_addr,
             rom_ctrl_reg_addr,
             uart_ctrl_reg_wr_data,
             rom_ctrl_reg_wr_data,
             reg_strt,
             reg_wr,
             reg_addr,
             reg_wr_data
             );

   input clk;
   input clk_en_2;
   input uart_ctrl_reg_strt;
   input rom_ctrl_reg_strt;
   input rom_ctrl_bsy;
   input uart_ctrl_reg_wr;
   input rom_ctrl_reg_wr;
   input [4:0] uart_ctrl_reg_addr;
   input [4:0] rom_ctrl_reg_addr;
   input [7:0] uart_ctrl_reg_wr_data;
   input [7:0] rom_ctrl_reg_wr_data;

   output reg reg_strt;          // synthesis attribute init of reg_strt is "0"
   output reg reg_wr;            // synthesis attribute init of reg_wr is "0"
   output reg [4:0] reg_addr;    // synthesis attribute init of reg_addr is "00000"
   output reg [7:0] reg_wr_data; // synthesis attribute init of reg_wr_data is "00000000"

   always @(posedge clk)
      if (clk_en_2)
         begin
            reg_strt <= rom_ctrl_bsy ? rom_ctrl_reg_strt : uart_ctrl_reg_strt;
            reg_wr <= rom_ctrl_bsy ? rom_ctrl_reg_wr : uart_ctrl_reg_wr;
            reg_addr <= rom_ctrl_bsy ? rom_ctrl_reg_addr : uart_ctrl_reg_addr;
            reg_wr_data <= rom_ctrl_bsy ? rom_ctrl_reg_wr_data : uart_ctrl_reg_wr_data;
         end

endmodule

module reg_rd_rom_wr(
                     clk,
                     clk_en_2,
                     strt,
                     reg_bsy,
                     rom_bsy,
                     reg_rd_data,
                     bsy,
                     reg_strt,
                     rom_strt,
                     rom_wr_data
                     );

   input clk;
   input clk_en_2;
   input strt;
   input reg_bsy;
```

```
   input rom_bsy;
   input [7:0] reg_rd_data;

   output reg bsy;               // synthesis attribute init of bsy is "0"
   output reg reg_strt;          // synthesis attribute init of reg_strt is "0"
   output reg rom_strt;          // synthesis attribute init of rom_strt is "0"
   output reg [7:0] rom_wr_data; // synthesis attribute init of rom_wr_data is "00000000"

   reg [4:0] state;              // synthesis attribute init of state is "00001"

   parameter IDLE       = 5'b00001;
   parameter REG_STRT   = 5'b00010;
   parameter REG_BUSY   = 5'b00100;
   parameter ROM_STRT   = 5'b01000;
   parameter ROM_BUSY   = 5'b10000;

   always @(posedge clk)
      if (clk_en_2)
         case (state)   // synthesis full_case
            IDLE:
               if (strt)
                  begin
                     bsy <= 1'b1;
                     reg_strt <= 1'b1;
                     state <= REG_STRT;
                  end
            REG_STRT:
               if (reg_bsy)
                  begin
                     reg_strt <= 1'b0;
                     state <= REG_BUSY;
                  end
            REG_BUSY:
               if (~reg_bsy)
                  begin
                     rom_strt <= 1'b1;
                     rom_wr_data <= reg_rd_data;
                     state <= ROM_STRT;
                  end
            ROM_STRT:
               if (rom_bsy)
                  begin
                     rom_strt <= 1'b0;
                     state <= ROM_BUSY;
                  end
            ROM_BUSY:
               if (~rom_bsy)
                  begin
                     bsy <= 1'b0;
                     state <= IDLE;
                  end
         endcase

endmodule

module rom(
           clk,
           clk_en_16,
           strt,
           wr,
           miso,
           addr,
           wr_data,
           bsy,
           sck,
           mosi,
           nss,
           rd_data
           );

   input clk;
```

```
   input clk_en_16;
   input strt;
   input wr;
   input miso;
   input [4:0] addr;
   input [7:0] wr_data;

   output nss;
   output mosi;

   output reg sck;
   output reg bsy;            // synthesis attribute init of bsy is "0"
   output reg [7:0] rd_data;

   reg init_strt;             // synthesis attribute init of init_strt is "0"
   reg rom_nss;               // synthesis attribute init of rom_nss is "1"
   reg rom_mosi;
   reg [4:0] count1;
   reg [17:0] count2;
   reg [24:0] state;          // synthesis attribute init of state is
"00000000000000000000000001"

   wire init_bsy;
   wire init_nss;
   wire init_mosi;

   parameter IDLE       = 25'b0000000000000000000000001;
   parameter INIT_STRT  = 25'b0000000000000000000000010;
   parameter INIT_BUSY  = 25'b0000000000000000000000100;
   parameter INST_BIT7  = 25'b0000000000000000000001000;
   parameter INST_BIT6  = 25'b0000000000000000000010000;
   parameter INST_BIT5  = 25'b0000000000000000000100000;
   parameter INST_BIT4  = 25'b0000000000000000001000000;
   parameter INST_BIT3  = 25'b0000000000000000010000000;
   parameter INST_BIT2  = 25'b0000000000000000100000000;
   parameter INST_BIT1  = 25'b0000000000000001000000000;
   parameter INST_BIT0  = 25'b0000000000000010000000000;
   parameter ADDR_BIT4  = 25'b0000000000000100000000000;
   parameter ADDR_BIT3  = 25'b0000000000001000000000000;
   parameter ADDR_BIT2  = 25'b0000000000010000000000000;
   parameter ADDR_BIT1  = 25'b0000000000100000000000000;
   parameter ADDR_BIT0  = 25'b0000000001000000000000000;
   parameter DATA_BIT7  = 25'b0000000010000000000000000;
   parameter DATA_BIT6  = 25'b0000000100000000000000000;
   parameter DATA_BIT5  = 25'b0000001000000000000000000;
   parameter DATA_BIT4  = 25'b0000010000000000000000000;
   parameter DATA_BIT3  = 25'b0000100000000000000000000;
   parameter DATA_BIT2  = 25'b0001000000000000000000000;
   parameter DATA_BIT1  = 25'b0010000000000000000000000;
   parameter DATA_BIT0  = 25'b0100000000000000000000000;
   parameter WAIT       = 25'b1000000000000000000000000;

   assign nss = init_bsy ? init_nss : rom_nss;
   assign mosi = init_bsy ? init_mosi : rom_mosi;

   rom_init rom_init(
                    .clk(clk),
                    .clk_en_16(clk_en_16),
                    .wr(wr),
                    .strt(init_strt),
                    .sck(sck),
                    .bsy(init_bsy),
                    .nss(init_nss),
                    .mosi(init_mosi)
                    );

   always @(posedge clk)
      if (clk_en_16)
         begin
            sck <= ~ sck;
            case (state)   // synthesis full_case
```

```
IDLE:
   if (strt)
      begin
         init_strt <= 1'b1;
         bsy <= 1'b1;
         state <= INIT_STRT;
      end
INIT_STRT:
   if (init_bsy)
      begin
         init_strt <= 1'b0;
         state <= INIT_BUSY;
      end
INIT_BUSY:
   if (sck & ~init_bsy)
      begin
         rom_nss <= 1'b0;
         rom_mosi <= 1'b0;
         state <= INST_BIT7;
      end
INST_BIT7:
   if (sck)
      state <= INST_BIT6;
INST_BIT6:
   if (sck)
      state <= INST_BIT5;
INST_BIT5:
   if (sck)
      state <= INST_BIT4;
INST_BIT4:
   if (sck)
      begin
         rom_mosi <= wr ? 1'b1 : 1'b0;
         state <= INST_BIT3;
      end
INST_BIT3:
   if (sck)
      begin
         rom_mosi <= 1'b0;
         state <= INST_BIT2;
      end
INST_BIT2:
   if (sck)
      begin
         rom_mosi <= 1'b1;
         state <= INST_BIT1;
      end
INST_BIT1:
   if (sck)
      begin
         rom_mosi <= wr ? 1'b0 : 1'b1;
         count1 <= 5'd19;
         state <= INST_BIT0;
      end
INST_BIT0:
   if (sck)
      begin
         count1 <= count1 - 5'd1;
         rom_mosi <= count1 ? 1'b0 : addr [4];
         state <= count1 ? state : ADDR_BIT4;
      end
ADDR_BIT4:
   if (sck)
      begin
         rom_mosi <= addr [3];
         state <= ADDR_BIT3;
      end
ADDR_BIT3:
   if (sck)
      begin
         rom_mosi <= addr [2];
```

```
            state <= ADDR_BIT2;
         end
ADDR_BIT2:
      if (sck)
         begin
            rom_mosi <= addr [1];
            state <= ADDR_BIT1;
         end
ADDR_BIT1:
      if (sck)
         begin
            rom_mosi <= addr [0];
            state <= ADDR_BIT0;
         end
ADDR_BIT0:
      if (sck)
         begin
            rom_mosi <= wr_data [7];
            state <= DATA_BIT7;
         end
DATA_BIT7:
      if (sck)
         begin
            rom_mosi <= wr_data [6];
            state <= DATA_BIT6;
         end
      else
         rd_data [7] <= miso;
DATA_BIT6:
      if (sck)
         begin
            rom_mosi <= wr_data [5];
            state <= DATA_BIT5;
         end
      else
         rd_data [6] <= miso;
DATA_BIT5:
      if (sck)
         begin
            rom_mosi <= wr_data [4];
            state <= DATA_BIT4;
         end
      else
         rd_data [5] <= miso;
DATA_BIT4:
      if (sck)
         begin
            rom_mosi <= wr_data [3];
            state <= DATA_BIT3;
         end
      else
         rd_data [4] <= miso;
DATA_BIT3:
      if (sck)
         begin
            rom_mosi <= wr_data [2];
            state <= DATA_BIT2;
         end
      else
         rd_data [3] <= miso;
DATA_BIT2:
      if (sck)
         begin
            rom_mosi <= wr_data [1];
            state <= DATA_BIT1;
         end
      else
         rd_data [2] <= miso;
DATA_BIT1:
      if (sck)
         begin
```

```
                        rom_mosi <= wr_data [0];
                        state <= DATA_BIT0;
                     end
                  else
                     rd_data [1] <= miso;
               DATA_BIT0:
                  if (sck)
                     begin
                        rom_nss <= 1'b1;
                        count2 <= 18'd0;
                        bsy <= wr ? 1'b1 : 1'b0;
                        state <= wr ? WAIT : IDLE;
                     end
                  else
                     rd_data [0] <= miso;
               WAIT:
                  if (count2 [17])
                     begin
                        bsy <= 1'b0;
                        state <= IDLE;
                     end
                  else
                     count2 <= count2 + 18'd1;
            endcase
         end

endmodule

module rom_ctrl(
                clk,
                clk_en_2,
                clk_en_16,
                miso,
                load,
                reg_bsy,
                reg_rd_data,
                bsy,
                nss,
                sck,
                mosi,
                reg_strt,
                reg_wr,
                reg_addr,
                reg_wr_data
                );

   input clk;
   input clk_en_2;
   input clk_en_16;
   input miso;
   input load;
   input reg_bsy;
   input [7:0] reg_rd_data;

   output nss;
   output sck;
   output mosi;
   output reg_strt;
   output [4:0] reg_addr;
   output [7:0] reg_wr_data;

   output reg bsy;         // synthesis attribute init of bsy is "1"
   output reg reg_wr;

   reg rom_wr;
   reg rom_rd_reg_wr_strt; // synthesis attribute init of rom_rd_reg_wr_strt is "1"
   reg reg_rd_rom_wr_strt; // synthesis attribute init of reg_rd_rom_wr_strt is "0"
   reg [4:0] addr;         // synthesis attribute init of addr is "00010"
   reg [72:0] state;       // synthesis attribute init of state is
"0000000000000000000000000000000000000000000000000000000000000000000000000000001"
```

```
   wire rom_bsy;
   wire rom_strt;
   wire rom_rd_reg_wr_reg_strt;
   wire reg_rd_rom_wr_reg_strt;
   wire rom_rd_reg_wr_rom_strt;
   wire reg_rd_rom_wr_rom_strt;
   wire rom_rd_reg_wr_bsy;
   wire reg_rd_rom_wr_bsy;
   wire [4:0] rom_addr;
   wire [7:0] rom_rd_data;
   wire [7:0] rom_wr_data;

   parameter REG_WR_STRT_2    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000000000001;
   parameter REG_WR_BUSY_2    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000000000010;
   parameter REG_WR_STRT_4    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000000000100;
   parameter REG_WR_BUSY_4    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000000001000;
   parameter REG_WR_STRT_5    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000000010000;
   parameter REG_WR_BUSY_5    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000000100000;
   parameter REG_WR_STRT_6    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000001000000;
   parameter REG_WR_BUSY_6    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000010000000;
   parameter REG_WR_STRT_7    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000100000000;
   parameter REG_WR_BUSY_7    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000001000000000;
   parameter REG_WR_STRT_8    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000010000000000;
   parameter REG_WR_BUSY_8    =
73'b0000000000000000000000000000000000000000000000000000000000000000000000100000000000;
   parameter REG_WR_STRT_9    =
73'b0000000000000000000000000000000000000000000000000000000000000000000001000000000000;
   parameter REG_WR_BUSY_9    =
73'b0000000000000000000000000000000000000000000000000000000000000000000010000000000000;
   parameter REG_WR_STRT_10   =
73'b0000000000000000000000000000000000000000000000000000000000000000000100000000000000;
   parameter REG_WR_BUSY_10   =
73'b0000000000000000000000000000000000000000000000000000000000000000001000000000000000;
   parameter REG_WR_STRT_11   =
73'b0000000000000000000000000000000000000000000000000000000000000000010000000000000000;
   parameter REG_WR_BUSY_11   =
73'b0000000000000000000000000000000000000000000000000000000000000000100000000000000000;
   parameter REG_WR_STRT_12   =
73'b0000000000000000000000000000000000000000000000000000000000000001000000000000000000;
   parameter REG_WR_BUSY_12   =
73'b0000000000000000000000000000000000000000000000000000000000000010000000000000000000;
   parameter REG_WR_STRT_13   =
73'b0000000000000000000000000000000000000000000000000000000000000100000000000000000000;
   parameter REG_WR_BUSY_13   =
73'b0000000000000000000000000000000000000000000000000000000000001000000000000000000000;
   parameter REG_WR_STRT_14   =
73'b0000000000000000000000000000000000000000000000000000000000010000000000000000000000;
   parameter REG_WR_BUSY_14   =
73'b0000000000000000000000000000000000000000000000000000000000100000000000000000000000;
   parameter REG_WR_STRT_15   =
73'b0000000000000000000000000000000000000000000000000000000001000000000000000000000000;
   parameter REG_WR_BUSY_15   =
73'b0000000000000000000000000000000000000000000000000000000010000000000000000000000000;
   parameter REG_WR_STRT_16   =
73'b0000000000000000000000000000000000000000000000000000000100000000000000000000000000;
   parameter REG_WR_BUSY_16   =
73'b0000000000000000000000000000000000000000000000000000001000000000000000000000000000;
   parameter REG_WR_STRT_17   =
73'b0000000000000000000000000000000000000000000000000000010000000000000000000000000000;
```

```
   parameter REG_WR_BUSY_17   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter REG_WR_STRT_20   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter REG_WR_BUSY_20   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter REG_WR_STRT_21   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter REG_WR_BUSY_21   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter REG_WR_STRT_22   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter REG_WR_BUSY_22   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter IDLE             =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_2    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_2    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_4    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_4    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_5    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_5    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_6    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_6    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_7    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_7    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_8    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_8    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_9    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_9    =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_10   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_10   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_11   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_11   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_12   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_12   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_13   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_13   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_14   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_14   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_15   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_BUSY_15   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
   parameter ROM_WR_STRT_16   =
73'b0000000000000000000000000000000000000000000010000000000000000000000000000000000;
```

```
   parameter ROM_WR_BUSY_16   =
73'b0000000010000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_STRT_17   =
73'b0000000100000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_BUSY_17   =
73'b0000001000000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_STRT_20   =
73'b0000010000000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_BUSY_20   =
73'b0000100000000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_STRT_21   =
73'b0001000000000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_BUSY_21   =
73'b0010000000000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_STRT_22   =
73'b0100000000000000000000000000000000000000000000000000000000000000000000000000000;
   parameter ROM_WR_BUSY_22   =
73'b1000000000000000000000000000000000000000000000000000000000000000000000000000000;

   assign reg_addr = addr;
   assign rom_addr = addr;
   assign reg_strt = rom_rd_reg_wr_bsy ? rom_rd_reg_wr_reg_strt : reg_rd_rom_wr_reg_strt;
   assign rom_strt = rom_rd_reg_wr_bsy ? rom_rd_reg_wr_rom_strt : reg_rd_rom_wr_rom_strt;

   rom rom(
           .clk(clk),
           .clk_en_16(clk_en_16),
           .strt(rom_strt),
           .wr(rom_wr),
           .miso(miso),
           .addr(rom_addr),
           .wr_data(rom_wr_data),
           .bsy(rom_bsy),
           .sck(sck),
           .mosi(mosi),
           .nss(nss),
           .rd_data(rom_rd_data)
           );

   rom_rd_reg_wr rom_rd_reg_wr(
                               .clk(clk),
                               .clk_en_2(clk_en_2),
                               .strt(rom_rd_reg_wr_strt),
                               .reg_bsy(reg_bsy),
                               .rom_bsy(rom_bsy),
                               .rom_rd_data(rom_rd_data),
                               .bsy(rom_rd_reg_wr_bsy),
                               .reg_strt(rom_rd_reg_wr_reg_strt),
                               .rom_strt(rom_rd_reg_wr_rom_strt),
                               .reg_wr_data(reg_wr_data)
                               );

   reg_rd_rom_wr reg_rd_rom_wr(
                               .clk(clk),
                               .clk_en_2(clk_en_2),
                               .strt(reg_rd_rom_wr_strt),
                               .reg_bsy(reg_bsy),
                               .rom_bsy(rom_bsy),
                               .reg_rd_data(reg_rd_data),
                               .bsy(reg_rd_rom_wr_bsy),
                               .reg_strt(reg_rd_rom_wr_reg_strt),
                               .rom_strt(reg_rd_rom_wr_rom_strt),
                               .rom_wr_data(rom_wr_data)
                               );

   always @(posedge clk)
      if (clk_en_2)
         begin
            case (state)   // synthesis full_case
               REG_WR_STRT_2:
                  begin
```

```
      reg_wr <= 1'b1;
      rom_wr <= 1'b0;
      if (rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b0;
            state <= REG_WR_BUSY_2;
         end
   end
REG_WR_BUSY_2:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd4;
         state <= REG_WR_STRT_4;
      end
REG_WR_STRT_4:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_4;
      end
REG_WR_BUSY_4:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd5;
         state <= REG_WR_STRT_5;
      end
REG_WR_STRT_5:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_5;
      end
REG_WR_BUSY_5:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd6;
         state <= REG_WR_STRT_6;
      end
REG_WR_STRT_6:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_6;
      end
REG_WR_BUSY_6:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd7;
         state <= REG_WR_STRT_7;
      end
REG_WR_STRT_7:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_7;
      end
REG_WR_BUSY_7:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd8;
         state <= REG_WR_STRT_8;
      end
REG_WR_STRT_8:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
```

```
            state <= REG_WR_BUSY_8;
         end
   REG_WR_BUSY_8:
      if (~rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b1;
            addr <= 5'd9;
            state <= REG_WR_STRT_9;
         end
   REG_WR_STRT_9:
      if (rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b0;
            state <= REG_WR_BUSY_9;
         end
   REG_WR_BUSY_9:
      if (~rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b1;
            addr <= 5'd10;
            state <= REG_WR_STRT_10;
         end
   REG_WR_STRT_10:
      if (rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b0;
            state <= REG_WR_BUSY_10;
         end
   REG_WR_BUSY_10:
      if (~rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b1;
            addr <= 5'd11;
            state <= REG_WR_STRT_11;
         end
   REG_WR_STRT_11:
      if (rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b0;
            state <= REG_WR_BUSY_11;
         end
   REG_WR_BUSY_11:
      if (~rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b1;
            addr <= 5'd12;
            state <= REG_WR_STRT_12;
         end
   REG_WR_STRT_12:
      if (rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b0;
            state <= REG_WR_BUSY_12;
         end
   REG_WR_BUSY_12:
      if (~rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b1;
            addr <= 5'd13;
            state <= REG_WR_STRT_13;
         end
   REG_WR_STRT_13:
      if (rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b0;
            state <= REG_WR_BUSY_13;
         end
   REG_WR_BUSY_13:
      if (~rom_rd_reg_wr_bsy)
         begin
            rom_rd_reg_wr_strt <= 1'b1;
```

```
         addr <= 5'd14;
         state <= REG_WR_STRT_14;
      end
REG_WR_STRT_14:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_14;
      end
REG_WR_BUSY_14:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd15;
         state <= REG_WR_STRT_15;
      end
REG_WR_STRT_15:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_15;
      end
REG_WR_BUSY_15:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd16;
         state <= REG_WR_STRT_16;
      end
REG_WR_STRT_16:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_16;
      end
REG_WR_BUSY_16:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd17;
         state <= REG_WR_STRT_17;
      end
REG_WR_STRT_17:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_17;
      end
REG_WR_BUSY_17:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd20;
         state <= REG_WR_STRT_20;
      end
REG_WR_STRT_20:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_20;
      end
REG_WR_BUSY_20:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd21;
         state <= REG_WR_STRT_21;
      end
REG_WR_STRT_21:
   if (rom_rd_reg_wr_bsy)
      begin
```

```
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_21;
      end
REG_WR_BUSY_21:
   if (~rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b1;
         addr <= 5'd22;
         state <= REG_WR_STRT_22;
      end
REG_WR_STRT_22:
   if (rom_rd_reg_wr_bsy)
      begin
         rom_rd_reg_wr_strt <= 1'b0;
         state <= REG_WR_BUSY_22;
      end
REG_WR_BUSY_22:
   if (~rom_rd_reg_wr_bsy)
      begin
         reg_wr <= 1'b0;
         rom_wr <= 1'b1;
         bsy <= 1'b0;
         state <= IDLE;
      end
IDLE:
   if (load)
      begin
         addr <= 5'd2;
         reg_rd_rom_wr_strt <= 1'b1;
         bsy <= 1'b1;
         state <= ROM_WR_STRT_2;
      end
ROM_WR_STRT_2:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_2;
      end
ROM_WR_BUSY_2:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd4;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_4;
      end
ROM_WR_STRT_4:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_4;
      end
ROM_WR_BUSY_4:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd5;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_5;
      end
ROM_WR_STRT_5:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_5;
      end
ROM_WR_BUSY_5:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd6;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_6;
      end
```

```
ROM_WR_STRT_6:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_6;
      end
ROM_WR_BUSY_6:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd7;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_7;
      end
ROM_WR_STRT_7:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_7;
      end
ROM_WR_BUSY_7:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd8;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_8;
      end
ROM_WR_STRT_8:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_8;
      end
ROM_WR_BUSY_8:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd9;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_9;
      end
ROM_WR_STRT_9:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_9;
      end
ROM_WR_BUSY_9:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd10;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_10;
      end
ROM_WR_STRT_10:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_10;
      end
ROM_WR_BUSY_10:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd11;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_11;
      end
ROM_WR_STRT_11:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_11;
      end
```

```
ROM_WR_BUSY_11:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd12;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_12;
      end
ROM_WR_STRT_12:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_12;
      end
ROM_WR_BUSY_12:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd13;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_13;
      end
ROM_WR_STRT_13:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_13;
      end
ROM_WR_BUSY_13:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd14;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_14;
      end
ROM_WR_STRT_14:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_14;
      end
ROM_WR_BUSY_14:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd15;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_15;
      end
ROM_WR_STRT_15:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_15;
      end
ROM_WR_BUSY_15:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd16;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_16;
      end
ROM_WR_STRT_16:
   if (reg_rd_rom_wr_bsy)
      begin
         reg_rd_rom_wr_strt <= 1'b0;
         state <= ROM_WR_BUSY_16;
      end
ROM_WR_BUSY_16:
   if (~reg_rd_rom_wr_bsy)
      begin
         addr <= 5'd17;
         reg_rd_rom_wr_strt <= 1'b1;
         state <= ROM_WR_STRT_17;
```

```
                end
            ROM_WR_STRT_17:
               if (reg_rd_rom_wr_bsy)
                  begin
                     reg_rd_rom_wr_strt <= 1'b0;
                     state <= ROM_WR_BUSY_17;
                  end
            ROM_WR_BUSY_17:
               if (~reg_rd_rom_wr_bsy)
                  begin
                     addr <= 5'd20;
                     reg_rd_rom_wr_strt <= 1'b1;
                     state <= ROM_WR_STRT_20;
                  end
            ROM_WR_STRT_20:
               if (reg_rd_rom_wr_bsy)
                  begin
                     reg_rd_rom_wr_strt <= 1'b0;
                     state <= ROM_WR_BUSY_20;
                  end
            ROM_WR_BUSY_20:
               if (~reg_rd_rom_wr_bsy)
                  begin
                     addr <= 5'd21;
                     reg_rd_rom_wr_strt <= 1'b1;
                     state <= ROM_WR_STRT_21;
                  end
            ROM_WR_STRT_21:
               if (reg_rd_rom_wr_bsy)
                  begin
                     reg_rd_rom_wr_strt <= 1'b0;
                     state <= ROM_WR_BUSY_21;
                  end
            ROM_WR_BUSY_21:
               if (~reg_rd_rom_wr_bsy)
                  begin
                     addr <= 5'd22;
                     reg_rd_rom_wr_strt <= 1'b1;
                     state <= ROM_WR_STRT_22;
                  end
            ROM_WR_STRT_22:
               if (reg_rd_rom_wr_bsy)
                  begin
                     reg_rd_rom_wr_strt <= 1'b0;
                     state <= ROM_WR_BUSY_22;
                  end
            ROM_WR_BUSY_22:
               if (~reg_rd_rom_wr_bsy)
                  begin
                     bsy <= 1'b0;
                     state <= IDLE;
                  end
         endcase
      end

endmodule

module rom_init(
               clk,
               clk_en_16,
               wr,
               strt,
               sck,
               bsy,
               nss,
               mosi
               );

   input clk;
   input clk_en_16;
   input wr;
```

```
   input strt;
   input sck;

   output reg bsy;   // synthesis attribute init of bsy is "0"
   output reg nss;   // synthesis attribute init of nss is "1"
   output reg mosi;  // synthesis attribute init of mosi is "0"

   reg [8:0] state;  // synthesis attribute init of state is "000000001"

   parameter IDLE = 9'b000000001;
   parameter BIT7 = 9'b000000010;
   parameter BIT6 = 9'b000000100;
   parameter BIT5 = 9'b000001000;
   parameter BIT4 = 9'b000010000;
   parameter BIT3 = 9'b000100000;
   parameter BIT2 = 9'b001000000;
   parameter BIT1 = 9'b010000000;
   parameter BIT0 = 9'b100000000;

   always @(posedge clk)
      if (clk_en_16 & sck)
         case (state)   // synthesis full_case
            IDLE:
               if (strt)
                  begin
                     bsy <= 1'b1;
                     nss <= 1'b0;
                     state <= BIT7;
                  end
            BIT7:
               state <= BIT6;
            BIT6:
               state <= BIT5;
            BIT5:
               state <= BIT4;
            BIT4:
               state <= BIT3;
            BIT3:
               begin
                  mosi <= 1'b1;
                  state <= BIT2;
               end
            BIT2:
               begin
                  mosi <= wr ? 1'b1 : 1'b0;
                  state <= BIT1;
               end
            BIT1:
               begin
                  mosi <= 1'b0;
                  state <= BIT0;
               end
            BIT0:
               begin
                  bsy <= 1'b0;
                  nss <= 1'b1;
                  state <= IDLE;
               end
         endcase

endmodule

module rom_rd_reg_wr(
                     clk,
                     clk_en_2,
                     strt,
                     reg_bsy,
                     rom_bsy,
                     rom_rd_data,
                     bsy,
                     reg_strt,
```

```
                    rom_strt,
                    reg_wr_data
                    );

   input clk;
   input clk_en_2;
   input strt;
   input reg_bsy;
   input rom_bsy;
   input [7:0] rom_rd_data;

   output reg bsy;               // synthesis attribute init of bsy is "0"
   output reg reg_strt;          // synthesis attribute init of reg_strt is "0"
   output reg rom_strt;          // synthesis attribute init of rom_strt is "0"
   output reg [7:0] reg_wr_data; // synthesis attribute init of reg_wr_data is "00000000"

   reg [4:0] state;              // synthesis attribute init of state is "00001"

   parameter IDLE       = 5'b00001;
   parameter ROM_STRT   = 5'b00010;
   parameter ROM_BUSY   = 5'b00100;
   parameter REG_STRT   = 5'b01000;
   parameter REG_BUSY   = 5'b10000;

   always @(posedge clk)
      if (clk_en_2)
         begin
            case (state)   // synthesis full_case
               IDLE:
                  if (strt)
                     begin
                        bsy <= 1'b1;
                        rom_strt <= 1'b1;
                        state <= ROM_STRT;
                     end
               ROM_STRT:
                  if (rom_bsy)
                     begin
                        rom_strt <= 1'b0;
                        state <= ROM_BUSY;
                     end
               ROM_BUSY:
                  if (~rom_bsy)
                     begin
                        reg_wr_data <= rom_rd_data;
                        reg_strt <= 1'b1;
                        state <= REG_STRT;
                     end
               REG_STRT:
                  if (reg_bsy)
                     begin
                        reg_strt <= 1'b0;
                        state <= REG_BUSY;
                     end
               REG_BUSY:
                  if (~reg_bsy)
                     begin
                        bsy <= 1'b0;
                        state <= IDLE;
                     end
            endcase
         end

endmodule

module rx(
          clk,
          baud_en_1,
          din,
          rset,
          full,
```

```
       dout
       );

input clk;
input baud_en_1;
input din;
input rset;

output reg full;           // synthesis attribute init of full is "0"
output reg [7:0] dout;     // synthesis attribute init of dout is "00000000"

reg din1;                  // synthesis attribute init of din1 is "1"
reg din2;                  // synthesis attribute init of din2 is "1"
reg [3:0] count;           // synthesis attribute init of count is "0000"
reg [11:0] state;          // synthesis attribute init of state is "000000000001"

parameter IDLE = 12'b000000000001;
parameter STRT = 12'b000000000010;
parameter BIT0 = 12'b000000000100;
parameter BIT1 = 12'b000000001000;
parameter BIT2 = 12'b000000010000;
parameter BIT3 = 12'b000000100000;
parameter BIT4 = 12'b000001000000;
parameter BIT5 = 12'b000010000000;
parameter BIT6 = 12'b000100000000;
parameter BIT7 = 12'b001000000000;
parameter STOP = 12'b010000000000;
parameter RSET = 12'b100000000000;

always @(posedge clk)
   begin
      if (baud_en_1)
         begin
            din1 <= din;
            din2 <= din1;
            count <= count + 4'd1;
            case (state)   // synthesis full_case
               IDLE:
                  begin
                     count <= 4'd0;
                     state <= din2 ? IDLE : STRT;
                  end
               STRT:
                  if (count == 4'd7)
                     begin
                        count <= 4'd0;
                        state <= BIT0;
                     end
               BIT0:
                  if (count == 4'd15)
                     begin
                        dout [0] <= din2;
                        count <= 4'd0;
                        state <= BIT1;
                     end
               BIT1:
                  if (count == 4'd15)
                     begin
                        dout [1] <= din2;
                        count <= 4'd0;
                        state <= BIT2;
                     end
               BIT2:
                  if (count == 4'd15)
                     begin
                        dout [2] <= din2;
                        count <= 4'd0;
                        state <= BIT3;
                     end
               BIT3:
                  if (count == 4'd15)
```

```
                        begin
                           dout [3] <= din2;
                           count <= 4'd0;
                           state <= BIT4;
                        end
                     BIT4:
                        if (count == 4'd15)
                        begin
                           dout [4] <= din2;
                           count <= 4'd0;
                           state <= BIT5;
                        end
                     BIT5:
                        if (count == 4'd15)
                        begin
                           dout [5] <= din2;
                           count <= 4'd0;
                           state <= BIT6;
                        end
                     BIT6:
                        if (count == 4'd15)
                        begin
                           dout [6] <= din2;
                           count <= 4'd0;
                           state <= BIT7;
                        end
                     BIT7:
                        if (count == 4'd15)
                        begin
                           dout [7] <= din2;
                           count <= 4'd0;
                           state <= STOP;
                        end
                     STOP:
                        if (count == 4'd15)
                        begin
                           full <= 1'b1;
                           count <= 4'd0;
                           state <= RSET;
                        end
                     RSET:
                        if (rset)
                        begin
                           full <= 1'b0;
                           state <= IDLE;
                        end
                  endcase
               end
      end

endmodule

module tdr(
           clk_p,
           clk_n,
           usb_rx,
           usb_tx,
           ram_clk_out,
           ram_cke,
           ram_ncs,
           ram_nras,
           ram_ncas,
           ram_nwe,
           ram_dqm,
           ram_badr,
           ram_adr_out,
           ram_dq,
           cf_data,
           cf_adr,
           ncs0,
           ncs1,
```

```
        ndmack,
        nreset,
        niord,
        niowr,
        iordy,
        puls,
        len,
        dlay,
        adc_data,
        miso,
        sck,
        mosi,
        rom_nss,
        adc_nss,
        dac_nss,
        rom_nrset,
        rom_nwp,
        atrg,
        sel,
        tmpgd,
        fin,
        tmpen
        );

input clk_p;
input clk_n;
input usb_rx;
input iordy;
input miso;
input atrg;
input tmpgd;
input fin;
input [11:0] adc_data;

inout [15:0] cf_data;
inout [23:0] ram_dq;

output sck;
output mosi;
output rom_nrset;
output rom_nwp;
output rom_nss;
output adc_nss;
output dac_nss;
output usb_tx;
output ram_clk_out;
output ram_cke;
output ram_ncs;
output ram_nras;
output ram_ncas;
output ram_nwe;
output ram_dqm;
output ncs0;
output ncs1;
output ndmack;
output nreset;
output niord;
output niowr;
output puls;
output len;
output sel;
output tmpen;
output [1:0] ram_badr;
output [2:0] cf_adr;
output [9:0] dlay;
output [12:0] ram_adr_out;

wire clk_en_2;
wire clk_en_4;
wire clk_en_16;
wire rate;
```

```
wire invrt;
wire buf_rd;
wire rx_full;
wire rx_rset;
wire reg_wr_en;
wire pio_wr_en;
wire acq_wr_en;
wire tx_strt;
wire acq_strt;
wire buf_strt;
wire ram_strt;
wire tx_bsy;
wire acq_bsy;
wire buf_bsy;
wire ram_bsy;
wire pio_bsy;
wire buf_ful;
wire buf_nxt;
wire load;
wire rom_ctrl_bsy;
wire rom_ctrl_reg_wr_en;
wire uart_ctrl_reg_wr_en;
wire reg_strt;
wire uart_ctrl_reg_strt;
wire rom_ctrl_reg_strt;
wire reg_bsy;
wire armd;
wire disarm;
wire fstrt;
wire fbsy;
wire [2:0] mode;
wire [1:0] step;
wire [2:0] pio_mode;
wire [4:0] reg_addr;
wire [4:0] rom_ctrl_reg_addr;
wire [4:0] uart_ctrl_reg_addr;
wire [5:0] attn;
wire [7:0] rx_data;
wire [7:0] tx_data;
wire [7:0] reg_rd_data;
wire [7:0] reg_wr_data;
wire [7:0] rom_ctrl_reg_wr_data;
wire [7:0] uart_ctrl_reg_wr_data;
wire [7:0] wdth;
wire [7:0] sampls;
wire [7:0] adc_dlay;
wire [7:0] acq_on;
wire [7:0] acq_off;
wire [9:0] dval;
wire [10:0] blk_num;
wire [11:0] buf_wrd_num;
wire [19:0] fval;
wire [23:0] pio_wr_data;
wire [23:0] acq_data;
wire [23:0] buf_rd_data;
wire [27:0] lba_strt;

IBUFGDS clkbuf(.O(clk), .I(clk_p), .IB(clk_n));
defparam clkbuf.IOSTANDARD = "LVDS_25";

assign adc_nss = 1'b1;
assign dac_nss = 1'b1;
assign rom_nwp = 1'b1;
assign rom_nrset = 1'b1;
assign rate = 1'b1;
assign sel = mode [2];

clk_en clk_en(
              .clk(clk),
              .clk_en_2(clk_en_2),
              .clk_en_4(clk_en_4),
```

```
              .clk_en_16(clk_en_16)
              );

uart uart(
          .clk(clk),
          .clk_en_4(clk_en_4),
          .rate(rate),
          .rx_din(usb_rx),
          .rx_rset(rx_rset),
          .rx_full(rx_full),
          .rx_dout(rx_data),
          .tx_strt(tx_strt),
          .tx_din(tx_data),
          .tx_bsy(tx_bsy),
          .tx_dout(usb_tx)
          );

uart_ctrl uart_ctrl(
                    .clk(clk),
                    .clk_en_2(clk_en_2),
                    .rx_full(rx_full),
                    .tx_bsy(tx_bsy),
                    .rx_data(rx_data),
                    .rd_data(reg_rd_data),
                    .wrd_num(buf_wrd_num),
                    .buf_data(buf_rd_data),
                    .buf_rd(buf_rd),
                    .wr_en(uart_ctrl_reg_wr_en),
                    .rx_rset(rx_rset),
                    .tx_strt(tx_strt),
                    .addr(uart_ctrl_reg_addr),
                    .wr_data(uart_ctrl_reg_wr_data),
                    .tx_data(tx_data),
                    .reg_strt(uart_ctrl_reg_strt),
                    .reg_bsy(reg_bsy)
                    );

reg_bank reg_bank(
                  .clk(clk),
                  .clk_en_2(clk_en_2),
                  .wr_en(reg_wr_en),
                  .addr(reg_addr),
                  .wr_data(reg_wr_data),
                  .rd_data(reg_rd_data),
                  .invrt(invrt),
                  .mode(mode),
                  .acq_strt(acq_strt),
                  .blk(blk_num),
                  .lba(lba_strt),
                  .acq_bsy(acq_bsy),
                  .pio_mode(pio_mode),
                  .step(step),
                  .wdth(wdth),
                  .sampls(sampls),
                  .adc_dlay(adc_dlay),
                  .buf_wrd_num(buf_wrd_num),
                  .attn(attn),
                  .rom_ctrl_bsy(rom_ctrl_bsy),
                  .load(load),
                  .reg_strt(reg_strt),
                  .reg_bsy(reg_bsy),
                  .acq_on(acq_on),
                  .acq_off(acq_off),
                  .armd(armd),
                  .disarm(disarm),
                  .tmpgd(tmpgd),
                  .dval(dval),
                  .fstrt(fstrt),
                  .fbsy(fbsy),
                  .fval(fval),
                  .tmpen(tmpen)
```

```
                  );

reg_mux reg_mux(
                .clk(clk),
                .clk_en_2(clk_en_2),
                .rom_ctrl_bsy(rom_ctrl_bsy),
                .uart_ctrl_reg_strt(uart_ctrl_reg_strt),
                .rom_ctrl_reg_strt(rom_ctrl_reg_strt),
                .uart_ctrl_reg_wr(uart_ctrl_reg_wr_en),
                .rom_ctrl_reg_wr(rom_ctrl_reg_wr_en),
                .uart_ctrl_reg_addr(uart_ctrl_reg_addr),
                .rom_ctrl_reg_addr(rom_ctrl_reg_addr),
                .uart_ctrl_reg_wr_data(uart_ctrl_reg_wr_data),
                .rom_ctrl_reg_wr_data(rom_ctrl_reg_wr_data),
                .reg_strt(reg_strt),
                .reg_wr(reg_wr_en),
                .reg_addr(reg_addr),
                .reg_wr_data(reg_wr_data)
                );

rom_ctrl rom_ctrl(
                  .clk(clk),
                  .clk_en_2(clk_en_2),
                  .clk_en_16(clk_en_16),
                  .miso(miso),
                  .load(load),
                  .reg_bsy(reg_bsy),
                  .reg_rd_data(reg_rd_data),
                  .bsy(rom_ctrl_bsy),
                  .nss(rom_nss),
                  .sck(sck),
                  .mosi(mosi),
                  .reg_strt(rom_ctrl_reg_strt),
                  .reg_wr(rom_ctrl_reg_wr_en),
                  .reg_addr(rom_ctrl_reg_addr),
                  .reg_wr_data(rom_ctrl_reg_wr_data)
                  );

acq_ctrl acq_ctrl(
                  .clk(clk),
                  .clk_en_2(clk_en_2),
                  .clk_en_4(clk_en_4),
                  .invrt(invrt),
                  .mode(mode),
                  .strt(acq_strt),
                  .buf_bsy(buf_bsy),
                  .ram_bsy(ram_bsy),
                  .step(step),
                  .wdth(wdth),
                  .sampls(sampls),
                  .adc_dlay(adc_dlay),
                  .buf_strt(buf_strt),
                  .ram_strt(ram_strt),
                  .wr_en(acq_wr_en),
                  .bsy(acq_bsy),
                  .puls(puls),
                  .len(len),
                  .dlay(dlay),
                  .adc_data(adc_data),
                  .ram_data(acq_data),
                  .acq_on(acq_on),
                  .acq_off(acq_off),
                  .dval(dval),
                  .atrg(atrg),
                  .armd(armd),
                  .disarm(disarm),
                  .pio_bsy(pio_bsy)
                  );

buff buff(
          .clk(clk),
```

```
            .clk_en_2(clk_en_2),
            .rd_en(buf_rd),
            .strt(buf_strt),
            .wr_en(acq_wr_en),
            .wrd_num(buf_wrd_num),
            .wr_data(acq_data),
            .bsy(buf_bsy),
            .rd_data(buf_rd_data)
            );

   ram_ctrl ram_ctrl(
                    .clk_in(clk),
                    .clk_en_2(clk_en_2),
                    .strt(ram_strt),
                    .wr_en(acq_wr_en),
                    .buf_nxt(buf_nxt),
                    .blk_num(blk_num),
                    .wr_data(acq_data),
                    .dq(ram_dq),
                    .clk_out(ram_clk_out),
                    .dqm(ram_dqm),
                    .cke(ram_cke),
                    .ncs(ram_ncs),
                    .nras(ram_nras),
                    .ncas(ram_ncas),
                    .nwe(ram_nwe),
                    .bsy(ram_bsy),
                    .pio_wr_en(pio_wr_en),
                    .buf_ful(buf_ful),
                    .badr(ram_badr),
                    .adr_out(ram_adr_out),
                    .rd_data(pio_wr_data)
                    );

   pio_ctrl pio_ctrl(
                    .clk(clk),
                    .clk_en_2(clk_en_2),
                    .iordy(iordy),
                    .wr_en(pio_wr_en),
                    .buf_ful(buf_ful),
                    .ram_bsy(ram_bsy),
                    .wr_data(pio_wr_data),
                    .lba_strt(lba_strt),
                    .cf_data(cf_data),
                    .ncs0(ncs0),
                    .ncs1(ncs1),
                    .ndmack(ndmack),
                    .nreset(nreset),
                    .niord(niord),
                    .niowr(niowr),
                    .buf_nxt(buf_nxt),
                    .cf_adr(cf_adr),
                    .mode(pio_mode),
                    .bsy(pio_bsy)
                    );

   countr countr(
                .clk(clk),
                .clk_en_16(clk_en_16),
                .fin(fin),
                .strt(fstrt),
                .bsy(fbsy),
                .dout(fval)
                );

endmodule

module tx(
          clk,
          baud_en_16,
          strt,
```

```
       din,
       bsy,
       dout
       );

input clk;
input baud_en_16;
input strt;
input [7:0] din;

output reg bsy;            // synthesis attribute init of bsy is "0"
output reg dout;           // synthesis attribute init of dout is "1"

reg [10:0] state;          // synthesis attribute init of state is "00000000001"

parameter IDLE = 11'b00000000001;
parameter STRT = 11'b00000000010;
parameter BIT0 = 11'b00000000100;
parameter BIT1 = 11'b00000001000;
parameter BIT2 = 11'b00000010000;
parameter BIT3 = 11'b00000100000;
parameter BIT4 = 11'b00001000000;
parameter BIT5 = 11'b00010000000;
parameter BIT6 = 11'b00100000000;
parameter BIT7 = 11'b01000000000;
parameter STOP = 11'b10000000000;

always @(posedge clk)
   begin
      if (baud_en_16)
         begin
            case (state)   // synthesis full_case
               IDLE:
                  if (strt)
                     begin
                        dout <= 1'b0;
                        bsy <= 1'b1;
                        state <= STRT;
                     end
               STRT:
                  begin
                     dout <= din [0];
                     state <= BIT0;
                  end
               BIT0:
                  begin
                     dout <= din [1];
                     state <= BIT1;
                  end
               BIT1:
                  begin
                     dout <= din [2];
                     state <= BIT2;
                  end
               BIT2:
                  begin
                     dout <= din [3];
                     state <= BIT3;
                  end
               BIT3:
                  begin
                     dout <= din [4];
                     state <= BIT4;
                  end
               BIT4:
                  begin
                     dout <= din [5];
                     state <= BIT5;
                  end
               BIT5:
                  begin
```

```
                        dout <= din [6];
                        state <= BIT6;
                     end
                  BIT6:
                     begin
                        dout <= din [7];
                        state <= BIT7;
                     end
                  BIT7:
                     begin
                        dout <= 1'b1;
                        state <= STOP;
                     end
                  STOP:
                     begin
                        bsy <= 1'b0;
                        state <= IDLE;
                     end
               endcase
            end
      end

endmodule

module uart(
            clk,
            clk_en_4,
            rate,
            rx_din,
            rx_rset,
            rx_full,
            rx_dout,
            tx_strt,
            tx_din,
            tx_bsy,
            tx_dout
            );

   input clk;
   input clk_en_4;
   input rate;
   input rx_din;
   input rx_rset;
   input tx_strt;
   input [7:0] tx_din;

   output rx_full;
   output tx_bsy;
   output tx_dout;
   output [7:0] rx_dout;

   wire baud_en_1;
   wire baud_en_16;

   baud_en baud_en(
                   .clk(clk),
                   .clk_en_4(clk_en_4),
                   .rate(rate),
                   .baud_en_1(baud_en_1),
                   .baud_en_16(baud_en_16)
                   );

   rx rx(
         .clk(clk),
         .baud_en_1(baud_en_1),
         .din(rx_din),
         .rset(rx_rset),
         .full(rx_full),
         .dout(rx_dout)
         );
```

```
   tx tx(
         .clk(clk),
         .baud_en_16(baud_en_16),
         .strt(tx_strt),
         .din(tx_din),
         .bsy(tx_bsy),
         .dout(tx_dout)
         );

endmodule

module uart_ctrl(
                clk,
                clk_en_2,
                rx_full,
                tx_bsy,
                rx_data,
                rd_data,
                wrd_num,
                buf_data,
                buf_rd,
                wr_en,
                rx_rset,
                tx_strt,
                addr,
                wr_data,
                tx_data,
                reg_strt,
                reg_bsy
                );

   input clk;
   input clk_en_2;
   input rx_full;
   input tx_bsy;
   input reg_bsy;
   input [7:0] rx_data;
   input [7:0] rd_data;
   input [11:0] wrd_num;
   input [23:0] buf_data;

   output reg reg_strt;       // synthesis attribute init of reg_strt is "0"
   output reg buf_rd;         // synthesis attribute init of buf_rd is "0"
   output reg wr_en;          // synthesis attribute init of wr_en is "0"
   output reg rx_rset;        // synthesis attribute init of rx_rset is "0"
   output reg tx_strt;        // synthesis attribute init of tx_strt is "0"
   output reg [4:0] addr;     // synthesis attribute init of addr is "00000"
   output reg [7:0] tx_data;  // synthesis attribute init of tx_data is "00000000"
   output reg [7:0] wr_data;  // synthesis attribute init of wr_data is "00000000"

   reg rd_buf;                // synthesis attribute init of rd_buf is "0"
   reg wr;                    // synthesis attribute init of wr is "0"
   reg [11:0] wrd_cnt;        // synthesis attribute init of wrd_cnt is "000000000000"
   reg [19:0] state;          // synthesis attribute init of state is
"00000000000000000001"

   parameter IDLE       = 20'b00000000000000000001;
   parameter RX_RSET_1  = 20'b00000000000000000010;
   parameter BUF_READ   = 20'b00000000000000000100;
   parameter D2_INIT    = 20'b00000000000000001000;
   parameter D2_STRT    = 20'b00000000000000010000;
   parameter D2_BUSY    = 20'b00000000000000100000;
   parameter D1_INIT    = 20'b00000000000001000000;
   parameter D1_STRT    = 20'b00000000000010000000;
   parameter D1_BUSY    = 20'b00000000000100000000;
   parameter D0_INIT    = 20'b00000000001000000000;
   parameter D0_STRT    = 20'b00000000010000000000;
   parameter D0_BUSY    = 20'b00000000100000000000;
   parameter READ_STRT  = 20'b00000001000000000000;
   parameter READ_BUSY  = 20'b00000010000000000000;
   parameter TX_STRT    = 20'b00000100000000000000;
```

```
parameter TX_BUSY    = 20'b00001000000000000000;
parameter WR_DATA    = 20'b00010000000000000000;
parameter RX_RSET_2  = 20'b00100000000000000000;
parameter WRITE_STRT = 20'b01000000000000000000;
parameter WRITE_BUSY = 20'b10000000000000000000;

always @(posedge clk)
   begin
      if (clk_en_2)
         begin
            case (state)   // synthesis full_case
               IDLE:
                  if (rx_full)
                     begin
                        rx_rset <= 1'b1;
                        addr <= rx_data [4:0];
                        wr <= rx_data [6];
                        rd_buf <= rx_data [7];
                        state <= RX_RSET_1;
                     end
               RX_RSET_1:
                  if (~rx_full)
                     begin
                        rx_rset <= 1'b0;
                        buf_rd <= rd_buf ? 1'b1 : buf_rd;
                        wrd_cnt <= wrd_num;
                        reg_strt <= ~(rd_buf | wr);
                        state <= rd_buf ? BUF_READ : wr ? WR_DATA : READ_STRT;
                     end
               BUF_READ:
                  begin
                     buf_rd <= 1'b0;
                     state <= D2_INIT;
                  end
               D2_INIT:
                  begin
                     tx_strt <= 1'b1;
                     tx_data <= buf_data [23:16];
                     state <= D2_STRT;
                  end
               D2_STRT:
                  if (tx_bsy)
                     begin
                        tx_strt <= 1'b0;
                        state <= D2_BUSY;
                     end
               D2_BUSY:
                  if (~tx_bsy)
                     state <= D1_INIT;
               D1_INIT:
                  begin
                     tx_strt <= 1'b1;
                     tx_data <= buf_data [15:8];
                     state <= D1_STRT;
                  end
               D1_STRT:
                  if (tx_bsy)
                     begin
                        tx_strt <= 1'b0;
                        state <= D1_BUSY;
                     end
               D1_BUSY:
                  if (~tx_bsy)
                     state <= D0_INIT;
               D0_INIT:
                  begin
                     tx_strt <= 1'b1;
                     tx_data <= buf_data [7:0];
                     state <= D0_STRT;
                  end
               D0_STRT:
```

```
                  if (tx_bsy)
                     begin
                        tx_strt <= 1'b0;
                        state <= D0_BUSY;
                     end
               D0_BUSY:
                  if (~tx_bsy)
                     begin
                        buf_rd <= wrd_cnt ? 1'b1 : buf_rd;
                        wrd_cnt <= wrd_cnt - 12'd1;
                        state <= wrd_cnt ? BUF_READ : IDLE;
                     end
               READ_STRT:
                  if (reg_bsy)
                     begin
                        reg_strt <= 1'b0;
                        state <= READ_BUSY;
                     end
               READ_BUSY:
                  if (~reg_bsy)
                     begin
                        tx_strt <= 1'b1;
                        tx_data <= rd_data;
                        state <= TX_STRT;
                     end
               TX_STRT:
                  if (tx_bsy)
                     begin
                        tx_strt <= 1'b0;
                        state <= TX_BUSY;
                     end
               TX_BUSY:
                  if (~tx_bsy)
                     state <= IDLE;
               WR_DATA:
                  if (rx_full)
                     begin
                        rx_rset <= 1'b1;
                        wr_data <= rx_data;
                        state <= RX_RSET_2;
                     end
               RX_RSET_2:
                  if (~rx_full)
                     begin
                        rx_rset <= 1'b0;
                        wr_en <= 1'b1;
                        reg_strt <= 1'b1;
                        state <= WRITE_STRT;
                     end
               WRITE_STRT:
                  if (reg_bsy)
                     begin
                        reg_strt <= 1'b0;
                        state <= WRITE_BUSY;
                     end
               WRITE_BUSY:
                  if (~reg_bsy)
                     begin
                        wr_en <= 1'b0;
                        state <= IDLE;
                     end
            endcase
         end
   end

endmodule

#Created by Constraints Editor (xc3s400-pq208-4) - 2010/07/03
NET "clk_p" TNM_NET = "clk_p";
TIMESPEC TS_clk_p = PERIOD "clk_p" 7 ns HIGH 50 %;
NET "clk_n" TNM_NET = "clk_n";
```

```
TIMESPEC TS_clk_n = PERIOD "clk_n" 7 ns HIGH 50 %;

# PlanAhead Generated physical constraints
NET "clk_p" LOC = P183;
NET "clk_n" LOC = P184;

# PlanAhead Generated physical constraints
# NET "usb_rx" LOC = P93;
# NET "usb_tx" LOC = P87;
NET "usb_rx" LOC = P79;
NET "usb_tx" LOC = P181;

# PlanAhead Generated IO constraints
NET "usb_rx" IOSTANDARD = LVCMOS33;
NET "usb_tx" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "ram_adr_out[0]" LOC = P24;
NET "ram_adr_out[1]" LOC = P26;
NET "ram_adr_out[2]" LOC = P27;
NET "ram_adr_out[3]" LOC = P28;
NET "ram_adr_out[4]" LOC = P29;
NET "ram_adr_out[5]" LOC = P31;
NET "ram_adr_out[6]" LOC = P33;
NET "ram_adr_out[7]" LOC = P34;
NET "ram_adr_out[8]" LOC = P35;
NET "ram_adr_out[9]" LOC = P36;
NET "ram_adr_out[10]" LOC = P22;
NET "ram_adr_out[11]" LOC = P37;
NET "ram_adr_out[12]" LOC = P39;
NET "ram_badr[0]" LOC = P20;
NET "ram_badr[1]" LOC = P21;
NET "ram_cke" LOC = P40;
NET "ram_clk_out" LOC = P42;
NET "ram_dq[0]" LOC = P63;
NET "ram_dq[1]" LOC = P62;
NET "ram_dq[2]" LOC = P61;
NET "ram_dq[3]" LOC = P58;
NET "ram_dq[4]" LOC = P57;
NET "ram_dq[5]" LOC = P52;
NET "ram_dq[6]" LOC = P51;
NET "ram_dq[7]" LOC = P50;
NET "ram_dq[8]" LOC = P48;
NET "ram_dq[9]" LOC = P46;
NET "ram_dq[10]" LOC = P45;
NET "ram_dq[11]" LOC = P44;
NET "ram_dq[12]" LOC = P13;
NET "ram_dq[13]" LOC = P12;
NET "ram_dq[14]" LOC = P11;
NET "ram_dq[15]" LOC = P10;
NET "ram_dq[16]" LOC = P9;
NET "ram_dq[17]" LOC = P7;
NET "ram_dq[18]" LOC = P5;
NET "ram_dq[19]" LOC = P4;
NET "ram_dq[20]" LOC = P3;
NET "ram_dq[21]" LOC = P2;
NET "ram_dq[22]" LOC = P205;
NET "ram_dq[23]" LOC = P204;
NET "ram_dqm" LOC = P43;
NET "ram_ncas" LOC = P16;
NET "ram_ncs" LOC = P19;
NET "ram_nras" LOC = P18;
NET "ram_nwe" LOC = P15;

# PlanAhead Generated IO constraints
NET "ram_adr_out[0]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[1]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[2]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[3]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[4]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[5]" IOSTANDARD = LVCMOS33;
```

```
NET "ram_adr_out[6]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[7]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[8]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[9]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[10]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[11]" IOSTANDARD = LVCMOS33;
NET "ram_adr_out[12]" IOSTANDARD = LVCMOS33;
NET "ram_badr[0]" IOSTANDARD = LVCMOS33;
NET "ram_badr[1]" IOSTANDARD = LVCMOS33;
NET "ram_cke" IOSTANDARD = LVCMOS33;
NET "ram_clk_out" IOSTANDARD = LVCMOS33;
NET "ram_dq[0]" IOSTANDARD = LVCMOS33;
NET "ram_dq[1]" IOSTANDARD = LVCMOS33;
NET "ram_dq[2]" IOSTANDARD = LVCMOS33;
NET "ram_dq[3]" IOSTANDARD = LVCMOS33;
NET "ram_dq[4]" IOSTANDARD = LVCMOS33;
NET "ram_dq[5]" IOSTANDARD = LVCMOS33;
NET "ram_dq[6]" IOSTANDARD = LVCMOS33;
NET "ram_dq[7]" IOSTANDARD = LVCMOS33;
NET "ram_dq[8]" IOSTANDARD = LVCMOS33;
NET "ram_dq[9]" IOSTANDARD = LVCMOS33;
NET "ram_dq[10]" IOSTANDARD = LVCMOS33;
NET "ram_dq[11]" IOSTANDARD = LVCMOS33;
NET "ram_dq[12]" IOSTANDARD = LVCMOS33;
NET "ram_dq[13]" IOSTANDARD = LVCMOS33;
NET "ram_dq[14]" IOSTANDARD = LVCMOS33;
NET "ram_dq[15]" IOSTANDARD = LVCMOS33;
NET "ram_dq[16]" IOSTANDARD = LVCMOS33;
NET "ram_dq[17]" IOSTANDARD = LVCMOS33;
NET "ram_dq[18]" IOSTANDARD = LVCMOS33;
NET "ram_dq[19]" IOSTANDARD = LVCMOS33;
NET "ram_dq[20]" IOSTANDARD = LVCMOS33;
NET "ram_dq[21]" IOSTANDARD = LVCMOS33;
NET "ram_dq[22]" IOSTANDARD = LVCMOS33;
NET "ram_dq[23]" IOSTANDARD = LVCMOS33;
NET "ram_dqm" IOSTANDARD = LVCMOS33;
NET "ram_ncas" IOSTANDARD = LVCMOS33;
NET "ram_ncs" IOSTANDARD = LVCMOS33;
NET "ram_nras" IOSTANDARD = LVCMOS33;
NET "ram_nwe" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "cf_adr[0]" LOC = P143;
NET "cf_adr[1]" LOC = P140;
NET "cf_adr[2]" LOC = P138;
NET "cf_data[0]" LOC = P146;
NET "cf_data[1]" LOC = P148;
NET "cf_data[2]" LOC = P150;
NET "cf_data[3]" LOC = P115;
NET "cf_data[4]" LOC = P117;
NET "cf_data[5]" LOC = P120;
NET "cf_data[6]" LOC = P123;
NET "cf_data[7]" LOC = P125;
NET "cf_data[8]" LOC = P149;
NET "cf_data[9]" LOC = P152;
NET "cf_data[10]" LOC = P155;
NET "cf_data[11]" LOC = P116;
NET "cf_data[12]" LOC = P119;
NET "cf_data[13]" LOC = P122;
NET "cf_data[14]" LOC = P124;
NET "cf_data[15]" LOC = P126;
NET "ncs0" LOC = P128;
NET "ncs1" LOC = P130;
NET "niord" LOC = P131;
NET "niowr" LOC = P132;

# PlanAhead Generated IO constraints
NET "cf_adr[0]" IOSTANDARD = LVCMOS33;
NET "cf_adr[1]" IOSTANDARD = LVCMOS33;
NET "cf_adr[2]" IOSTANDARD = LVCMOS33;
NET "cf_data[0]" IOSTANDARD = LVCMOS33;
```

```
NET "cf_data[1]" IOSTANDARD = LVCMOS33;
NET "cf_data[2]" IOSTANDARD = LVCMOS33;
NET "cf_data[3]" IOSTANDARD = LVCMOS33;
NET "cf_data[4]" IOSTANDARD = LVCMOS33;
NET "cf_data[5]" IOSTANDARD = LVCMOS33;
NET "cf_data[6]" IOSTANDARD = LVCMOS33;
NET "cf_data[7]" IOSTANDARD = LVCMOS33;
NET "cf_data[8]" IOSTANDARD = LVCMOS33;
NET "cf_data[9]" IOSTANDARD = LVCMOS33;
NET "cf_data[10]" IOSTANDARD = LVCMOS33;
NET "cf_data[11]" IOSTANDARD = LVCMOS33;
NET "cf_data[12]" IOSTANDARD = LVCMOS33;
NET "cf_data[13]" IOSTANDARD = LVCMOS33;
NET "cf_data[14]" IOSTANDARD = LVCMOS33;
NET "cf_data[15]" IOSTANDARD = LVCMOS33;
NET "ncs0" IOSTANDARD = LVCMOS33;
NET "ncs1" IOSTANDARD = LVCMOS33;
NET "niord" IOSTANDARD = LVCMOS33;
NET "niowr" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "iordy" LOC = P137;

# PlanAhead Generated IO constraints
NET "iordy" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "ndmack" LOC = P141;

# PlanAhead Generated IO constraints
NET "ndmack" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "nreset" LOC = P135;

# PlanAhead Generated IO constraints
NET "nreset" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "adc_data[0]" LOC = P182;
NET "adc_data[1]" LOC = P178;
NET "adc_data[2]" LOC = P176;
NET "adc_data[3]" LOC = P175;
NET "adc_data[4]" LOC = P172;
NET "adc_data[5]" LOC = P171;
NET "adc_data[6]" LOC = P169;
NET "adc_data[7]" LOC = P168;
NET "adc_data[8]" LOC = P167;
NET "adc_data[9]" LOC = P166;
NET "adc_data[10]" LOC = P165;
NET "adc_data[11]" LOC = P162;
NET "dlay[0]" LOC = P76;
NET "dlay[1]" LOC = P72;
NET "dlay[2]" LOC = P68;
NET "dlay[3]" LOC = P65;
NET "dlay[4]" LOC = P64;
NET "dlay[5]" LOC = P67;
NET "dlay[6]" LOC = P71;
NET "dlay[7]" LOC = P74;
NET "dlay[8]" LOC = P77;
NET "dlay[9]" LOC = P81;
NET "len" LOC = P185;
NET "puls" LOC = P187;

# PlanAhead Generated IO constraints
NET "adc_data[0]" IOSTANDARD = LVCMOS33;
NET "adc_data[1]" IOSTANDARD = LVCMOS33;
NET "adc_data[2]" IOSTANDARD = LVCMOS33;
NET "adc_data[3]" IOSTANDARD = LVCMOS33;
NET "adc_data[4]" IOSTANDARD = LVCMOS33;
NET "adc_data[5]" IOSTANDARD = LVCMOS33;
```

```
NET "adc_data[6]" IOSTANDARD = LVCMOS33;
NET "adc_data[7]" IOSTANDARD = LVCMOS33;
NET "adc_data[8]" IOSTANDARD = LVCMOS33;
NET "adc_data[9]" IOSTANDARD = LVCMOS33;
NET "adc_data[10]" IOSTANDARD = LVCMOS33;
NET "adc_data[11]" IOSTANDARD = LVCMOS33;
NET "dlay[0]" IOSTANDARD = LVCMOS33;
NET "dlay[1]" IOSTANDARD = LVCMOS33;
NET "dlay[2]" IOSTANDARD = LVCMOS33;
NET "dlay[3]" IOSTANDARD = LVCMOS33;
NET "dlay[4]" IOSTANDARD = LVCMOS33;
NET "dlay[5]" IOSTANDARD = LVCMOS33;
NET "dlay[6]" IOSTANDARD = LVCMOS33;
NET "dlay[7]" IOSTANDARD = LVCMOS33;
NET "dlay[8]" IOSTANDARD = LVCMOS33;
NET "dlay[9]" IOSTANDARD = LVCMOS33;
NET "len" IOSTANDARD = LVCMOS33;
NET "puls" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints
NET "adc_nss" LOC = P109;
NET "dac_nss" LOC = P97;
NET "miso" LOC = P107;
NET "mosi" LOC = P106;
NET "rom_nrset" LOC = P101;
NET "rom_nss" LOC = P100;
NET "rom_nwp" LOC = P108;
NET "sck" LOC = P102;

# PlanAhead Generated IO constraints
NET "adc_nss" IOSTANDARD = LVCMOS33;
NET "dac_nss" IOSTANDARD = LVCMOS33;
NET "miso" IOSTANDARD = LVCMOS33;
NET "mosi" IOSTANDARD = LVCMOS33;
NET "rom_nrset" IOSTANDARD = LVCMOS33;
NET "rom_nss" IOSTANDARD = LVCMOS33;
NET "rom_nwp" IOSTANDARD = LVCMOS33;
NET "sck" IOSTANDARD = LVCMOS33;
#Created by Constraints Editor (xc3s400-pq208-4) - 2010/11/06
TEMPERATURE  = 85 C;

# PlanAhead Generated physical constraints
NET "atrg" LOC = P80;

# PlanAhead Generated IO constraints
NET "atrg" IOSTANDARD = LVCMOS33;
NET "atrg" PULLDOWN;

# PlanAhead Generated physical constraints
NET "sel" LOC = P85;
NET "tmpgd" LOC = P78;

# PlanAhead Generated IO constraints
NET "sel" IOSTANDARD = LVCMOS33;
NET "tmpgd" IOSTANDARD = LVCMOS33;
#Created by Constraints Editor (xc3s400-pq208-4) - 2011/04/15
NET "fin" TNM_NET = "fin";
TIMESPEC TS_fin = PERIOD "fin" 8 ns HIGH 50 %;

# PlanAhead Generated physical constraints

NET "fin" LOC = P180;

# PlanAhead Generated IO constraints

NET "fin" IOSTANDARD = LVCMOS33;

# PlanAhead Generated physical constraints

NET "tmpen" LOC = P111;
```

```
# PlanAhead Generated IO constraints

NET "tmpen" IOSTANDARD = LVCMOS33;
```

BIBLIOGRAPHY

[1] H. M. Mu, "Development and Validation of Coaxial Cable Sensors for Damage Detection of Reinforced Concrete Structures," Ph.D. dissertation, Univ. Missouri-Rolla, Rolla, MO, 2003.

[2] S. Sun, D. J. Pommerenke, J. L. Drewniak, G. D. Chen, L. Xue, M. A. Brower, and M. Y. Koledintseva, "A Novel TDR-Based Coaxial Cable Sensor for Crack/Strain Sensing in Reinforced Concrete Structures," *IEEE Trans. Instrum. Meas.*, vol. 58, no. 8, pp. 2714–2725, Aug. 2009.

[3] G. Chen, B. Wood, J. Baird, I. Izyumin, and D. Pommerenke, Validation of Coaxial Cable Sensors for Dynamic Crack Detection in RC Columns Under Blast Loads, Jul. 4, 2008. [Online]. Available: http://www.dot.ca.gov/hq/esc/earthquake_ engineering/Research_Reports/vendor/missouri_s_and_t/CIES_08-74.pdf

[4] S. C. Burkhart and R. B. Wilcox, "Arbitrary pulse shape synthesis via nonuniform transmission lines," *IEEE Trans. Microw. Theory Tech.*, vol. 38, no. 10, pp. 1514-1518, Oct. 1990.

[5] D. J. Dascher, "Measuring Parasitic Capacitance and Inductance Using TDR," *Hewlett-Packard Journal*, vol. 47, no. 2, pp. 83-96, Apr. 1996.

[6] K. Schoen, "High Resolution TDR Measurements Using the PSPL Model 4020 and Model 4022 9 ps TDR Source Enhancement Modules," Picosecond Pulse Labs, Boulder, CO, July 2003.

[7] J. A. Bishop, D. J. Pommerenke, and G. Chen, "A Rapid-Acquisition Electrical Time-Domain Reflectometer for Dynamic Structure Analysis," *IEEE Trans. Instrum. Meas.*, vol. 60, no. 2, pp. 655-661, Feb. 2011.

[8] W. Maichen, "Digital Timing Measurements," Frontiers in Electronic Testing, vol. 33, pp. 217-221, Springer, 2006.

[9] J. R. Andrews, “Deconvolution of System Impulse Responses and Time Domain Waveforms,” Picosecond Pulse Labs, Boulder, CO, Oct. 2004.

VITA

Joseph Andrew Bishop was born on January 5, 1985 in Harrisburg, Illinois. He received the Bachelor of Science degree in electrical engineering from the Missouri University of Science and Technology in December of 2007. He received the Master of Science degree in electrical engineering from the Missouri University of Science and Technology in December of 2011.

Joseph interned with Sun Transformer in McLeansboro, Illinois during the summer of 2005. During the summer of 2006 he interned with Caterpillar in Peoria, Illinois. During the summer of 2007 he interned with Dynetics in Huntsville, Alabama. In January of 2008 he was hired into the Electromagnetic Compatibility Laboratory of the Missouri University of Science and Technology. During the summer of 2011 he interned with Agilent Technologies in Santa Rosa, California.

Joseph is a member of the Eta Kappa Nu honor society and served as Vice President of the Gamma Theta chapter during the spring of 2007. He is also a member of the Phi Eta Sigma, Tau Beta Pi, and Phi Kappa Phi honor societies.