Using Deep Serial Memory for Large Block Data Transfers

Tarra Epstein & Stephen Allen
System Test Group
Teradyne, Inc.
North Reading, MA
Tarra.Epstein@Teradyne.com
Stephen.Allen@Teradyne.com

Abstract— This paper will explain some of the advantages of utilizing an instrument with Deep Serial Memory in digital test applications that involve large block data transfer. Deep Serial Memory can accommodate test applications that use large block transfers of data, such as read-only memory (ROM) testing, boundary scan and communications that involve read/write of megawords of data. Conventional test pattern memory can be exhausted by these applications.

A test case will be presented to simplify the creation and support of tests for a Read Only Memory (ROM). The techniques applied in this example show how the transfer of large blocks of data can be easier and more practical with Deep Serial Memory.

Keywords-Deep Serial Memory; Memory Test; Throughput; Digital Test

I. INTRODUCTION

A digital test instrument uses Deep Serial Memory to store drive/detect data that can be utilized during digital test execution. This Deep Serial Memory is associated with every Digital Test Unit (DTU) pin and supports narrow or wide buses, considerably reducing the load time experienced with conventional memory. Deep Serial Memory is utilized to simplify digital test creation when transferring large blocks of data.

II. DEEP SERIAL MEMORY OVERVIEW

A. Advantages

Conventional pattern memory can be exhausted in applications that involve bulk transfers of data, such as ROM testing and communications that involve millions of read/writes. When Deep Serial Memory is associated with DTU pins, these applications can be easier to execute. A few basic patterns consisting of repetitive control signals can be loaded into conventional pattern memory and looped millions of times. The unique data for these cycles is stored in each channel’s Deep Serial Memory.

Another advantage in using Deep Serial Memory is the throughput advantage when compared with loading millions of conventional memory patterns. Pattern sequences that implement Deep Serial Memory consist of fewer patterns and load much faster than pattern sequences stored only in conventional pattern memory, improving the efficiency of large block data transfers. The actual time to load pattern sequences for all sections of a dynamic pattern set into pattern memory across the modules of a test instrument depends on the hardware and software configuration of the system.

B. Deep Serial Memory Data

The data stored in the Deep Serial Memory is interpreted based on its data mode. In a typical instrument design, four data modes provide the formatting options for data loaded into the Deep Serial Memory. The instrument’s Deep Serial Memory (also known as logic memory (LRAM)) has a depth of 8Mb in the OneBit and TwoBits modes and 4Mb in the OneDriveBitOneDetectBit and TwoDriveBitsTwoDetectBits modes. Fig. 1 lists the drive and expect states relating to the One-Bit mode or Two-Bit mode values loaded into Deep Serial Memory. OneDriveBitOneDetectBit mode uses a One-Bit value to select the Drive state and a second (least significant bit) One-Bit value to select the Detect state, with a total of two bits. The TwoDriveBitsTwoDetectBits mode uses two Two-Bit values to select Drive and Detect states, with a total of four bits.

<table>
<thead>
<tr>
<th>One-Bit Value</th>
<th>Two-Bit Value</th>
<th>Drive Opcode</th>
<th>Expect Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin0</td>
<td>bin00</td>
<td>Input low</td>
<td>Output low</td>
</tr>
<tr>
<td>bin1</td>
<td>bin01</td>
<td>Input high</td>
<td>Output high</td>
</tr>
<tr>
<td>N/A</td>
<td>bin10</td>
<td>Input between</td>
<td>Output between</td>
</tr>
<tr>
<td>N/A</td>
<td>bin11</td>
<td>Input Tristate</td>
<td>Output Don’t Care</td>
</tr>
</tbody>
</table>

Figure 1 - Opcode Data Format Table
C. Pattern Memory Opcodes

There are several opcodes that are stored in the pattern memory that cause channels to act on the data stored in the Deep Serial Memory. These opcodes are:

- EA (Expect Algorithmic)
- DA (Drive Algorithmic)
- IA (DriveOpcode.Algorithmic | ExpectOpcode.DontCare)
- OA (DriveOpcode.HiZ | ExpectOpcode.Algorithmic)
- MA (DriveOpcode.Algorithmic | ExpectOpcode.Algorithmic)

There is one pattern modifier to clock the data out of the Deep Serial Memory: ClockAlgorithmicEngine. The algorithmic engine is only advanced on those patterns for which the ClockAlgorithmicEngine modifier is specified. It will remain unchanged on all other patterns that do not call the modifier.

D. Pattern Data Organization

Implementing channels’ Deep Serial Memory allows for the separation of the Unit Under Test (UUT) protocol from the application of data. The protocol can be applied using conventional pattern memory, while the data to be transferred or tested is loaded into the Deep Serial Memory. The protocol patterns can then be reused independent of the data to be transferred or tested.

Millions of drive/detect cycles can be constructed of a few patterns in a loop repeated a million times. The data for each of the cycles is stored in Deep Serial Memory for each of the associated channels. The Deep Serial Memory can be advanced each time the data is accessed. Depicted in Fig. 2 and Fig. 3 are a comparison of how to program read and write cycles using conventional pattern memory and using Deep Serial Memory.

The creation and support of tests that involve large block transfers of data can be simplified with the use of Deep Serial Memory because of the reduced number of patterns.

III. Example Deep Serial Memory Application

A. ROM Testing

ROM (Read Only Memory) is a type of memory that can only be read and not written to. To test every location of ROM that contains millions of memory locations using conventional pattern memory, millions of patterns would need to be created. Deep Serial Memory makes ROM testing much simpler.

The test circuit for this example is a simple ROM with inputs of ReadEnable and Address (pins AddressPin19 through AddressPin0) and outputs of Data (DataPin19 through DataPin0). The address bus is a binary depiction of memory locations. Data from the address location is transferred to the data bus when the chip is active. Fig. 4 depicts a ROM test setup.

This example demonstrates only functional ROM testing, intended to detect permanent faults. The patterns presented here could be used to perform parametric testing, such as ROM voltage levels or timing at the input and output pins. The correct data expected at each memory location is known in advance, and the addresses are cycled through, first in ascending order and then descending order. This technique verifies that data is not destroyed by multiple reading actions.

This ROM test provides an example of how implementing Deep Serial Memory straightforwardly allows the separation of protocol and test data. The protocol patterns to set the Read Enable to an active low state can be executed with conventional pattern memory. Then the test patterns can source Address and Data pin drive/detect states from the Deep Serial Memory.
The ROM test data for this example is stored in an ASCII text file, which contains 40-bit hexadecimal words, approximately $2.1 \times 10^6$ deep. The least significant 20 bits of each word are associated with the ROM address bus, and bits 20 through 39 are associated with the data expected at that ROM location. The data in this file is read by a C# program and stored in an array to be loaded into an OpcodeDataTable, a table that holds data used as channel drive and expect opcodes.

The following lists sample ASCII hexadecimal data for this example:

```
CC1B500000
CC1B3000001
CC1B000002
.
00A0AFFFFE
00A0BFFFFF
00A0BFFFFF
00A0AFFFFE
.
.
CC1B000002
CC1B300001
CC1B500000
```

The data is loaded with the OneBit Opcode Data Format, meaning one bit value selects between drive high / drive low instructions for each pin (when used with the drive side of a channel) or expect high / expect low instructions for each pin (when used with the expect side of a channel). The 40 Address and Data bus pins are associated with the loaded opcode data when the Load function is called. This example uses pin groups to simplify Deep Serial Memory loading and pattern creation, but it is also possible to specify each pin individually when loading Deep Serial Memory or creating patterns.

The ROM circuit is set to a known state before testing begins, and the ROM is set to an active state through protocol patterns using conventional opcodes. The Address channel pins drive the data from the Deep Serial Memory associated with each pin, using the opcode DA. The Data channels expect based on the data from Deep Serial Memory associated with those pins, using the opcode EA. The ClockAlgorithmicEngine Pattern modifier is used for the test patterns to advance the Deep Serial Memory at the end of the pattern. Two patterns are looped until every data word stored in the Deep Serial Memory is tested. Note, a minimum of 2 patterns was looped for this test because the count limit for counted loops and repeats of this typical digital test instrument is 1,048,577, and the ROM has approximately $2.1 \times 10^6$ words of data to test.

To test ROM with more address locations, load an OpcodeDataTable with data for all channels the Deep Serial Memory is to be associated with. The Load function can only be called once or previously loaded data will be overwritten. A section of code to demonstrate the use of Deep Serial Memory for ROM functionality testing is provided, and an entire test program is outlined in Fig. 5:

```
/****** Create pin groups ******/
/* Create a pin group "Address" that will be used for the 20 address channel pins. */
Address = instrument.CreatePinGroup("Address",
AddressPin0, AddressPin1, AddressPin2, AddressPin3,
AddressPin4, AddressPin5, AddressPin6, AddressPin7,
AddressPin8, AddressPin9, AddressPin10, AddressPin11,
AddressPin12, AddressPin13, AddressPin14, AddressPin15,
AddressPin16, AddressPin17, AddressPin18, AddressPin19);

/* Create a pin group "Data" that will be used for the 20 data channel pins */
Data = instrument.CreatePinGroup("Data",
DataPin0, DataPin1, DataPin2, DataPin3, DataPin4,
DataPin5, DataPin6, DataPin7, DataPin8, DataPin9,
DataPin10, DataPin11, DataPin12, DataPin13, DataPin14,
DataPin15, DataPin16, DataPin17, DataPin18, DataPin19);
```

Figure 5 - ROM Test Code Flowchart
***** Read in test data from ASCII file *****

/* Implement a TextReader that reads characters from a byte stream in a particular encoding. User may need to change directory to locate the test file. Read the characters from an ASCII file “ROM_AddressData.txt” */
StreamReader ROMfile = new StreamReader(@"ROM_AddressData.txt");

/* Add each line from the ASCII file to the ArrayList AddressDataLines. Each line will be stored in the “stringLine” variable and added to the ArrayList. Each line is the 40-bit word we want to store in the Deep Serial Memory. */
ArrayList AddressDataLines = new ArrayList();
while((stringLine = ROMfile.ReadLine()) != null)
{
    AddressDataLines.Add(stringLine);
}

/* Convert the string representation of each Hex number to its 64-bit signed integer equivalent and store in AddressAndDataArray[] */
int DataLocation = 0;
AddressAndDataArray = new long [AddressDataLines.Count];
foreach (string String in AddressDataLines)
{
    long AddressAndDataLong = long.Parse(String, System.Globalization.NumberStyles.HexNumber);
    AddressAndDataArray[DataLocation] = AddressAndDataLong;
    DataLocation++;
}

/* Close the StreamReader object and the underlying stream, and releases any system resources associated with the reader. */
ROMfile.Close();

***** Load Opcode Data Table *****

/* Create an OpcodeDataTable associated with the Di session */
OpcodeDataTable opcodeDataTable = instrument.OpcodeDataTable;

/* Load the created OpcodeDataTable. Use OpcodeDataFormat OneBit to assign one bit to each channel. Load the AddressAndDataArray read from the ASCII file. Specify the pins to associate the OpcodeDataTable data with (in this case, pin groups) beginning with the pin to associate with the LSB. */
opcodeDataTable.Load(OpcodeDataFormat.OneBit, AddressAndDataArray, Address, Data);

***** Create Protocol patterns *****

/* Create a new Dynamic Pattern Set */
DynamicPatternSet dynamicPatternSet = instrument.PrimaryDynamicPatternSet;

/* Create a new Pattern Block */
PatternBlock patternBlock = new PatternBlock(instrument);

/* ROM circuit must be in a known state before executing Address/Data testing. Create pattern to set ReadEnable pin low (active). */
This pin will remain low for the entire pattern set execution if the Return Format in Nonreturn.
No test is performed on this pattern. */
patternBlock.PatternModifiers.TimingSet(0);
patternBlock.IL(ReadEnable);
patternBlock.IOX(Address, Data);
patternBlock.EndPattern(TestInstruction.None);

***** Create test patterns *****

/* First pattern uses the Loop control to begin the counted loop. Loop two patterns to test all data loaded into the Deep Serial memory. The address pins drive the opcode Deep Serial data associated with the pins. The data pins expect the opcode Deep Serial data associated with the pins. The Algorithmic Engine is clocked (advanced) at the end of the pattern. */
patternBlock.DA(Address);
patternBlock.EA(Data);
patternBlock.PatternModifiers.ClockAlgorithmicEngine();
patternBlock.PatternControl.Loop((AddressAndDataArray.Length/2));
patternBlock.EndPattern(TestInstruction.PassFail);

/* Second pattern uses the EndLoop control to end the counted loop. Since the first and second patterns clock the algorithmic engine, all data will be tested by the loop. The address pins drive the opcode Deep Serial data associated with the pins. The data pins expect the opcode Deep Serial data associated with the pins. The Algorithmic Engine is clocked (advanced) at the end of the pattern. */
patternBlock.DA(Address);
patternBlock.EA(Data);
patternBlock.PatternModifiers.ClockAlgorithmicEngine();
patternBlock.PatternControl.EndLoop();
patternBlock.EndPattern(TestInstruction.PassFail);

/* Last pattern uses the Halt control to stop the pattern set execution. All pins are tristated. No test is performed on this pattern. */
patternBlock.IOX(Address, Data);
patternBlock.PatternModifiers.TimingSet(0);
patternBlock.PatternControl.Halt(PatcCondition.True);
patternBlock.EndPattern(TestInstruction.None);

***** Load and Execute pattern set *****

/* Load the pattern data contained in the specified pattern sequences into pattern memory. Any existing pattern memory consumed by this dynamic pattern set is freed before loading the new pattern data. */
dynamicPatternSet.Load(patternBlock);

/* Initiate execution of the dynamic pattern set, wait for it to complete, and then return the test result in the variable “testResult”. */
testResult = dynamicPatternSet.Execute();
IV. RESULTS AND CONCLUSIONS

Deep Serial Memory benefits test programs that transfer large blocks of data. Implementing Deep Serial Memory simplifies the creation and support of tests with substantial bus communications and tests such as ROM location testing. The programming benefits of implementing Deep Serial Memory can be seen in the code example provided in this paper. The example would have required multiple pattern sets if constructed only using conventional pattern memory, which has a typical limit of 256k patterns.

Another important benefit of Deep Serial Memory use is the throughput advantage when loading pattern sets into hardware. The pattern set loading time is compared in the chart (Fig.6) for loading conventional pattern memory sets versus loading the Deep Serial Memory and a single pattern set. These results were obtained by executing tests using the ROM example setup of 40 test pins and do not include execution time, which would not be reduced by implementing Deep Serial Memory if the timing of each pattern set was not changed.

![Deep Serial Memory vs. Conventional Pattern Set Load Times](chart.png)

**Figure 6 - Pattern Set Load Time Comparison Chart**

This chart shows that the time to load multiple pattern sets using only conventional memory versus loading Deep Serial Memory and a single pattern set is approximately 10:1. The results reported in the chart, however, are specific to the system configuration being used and may vary slightly with a different configuration. The configuration used to obtain these data points is described below:

- Windows 2000 SP4 Operating System
- 3.2 GHZ Processor
- 1G RAM
- NI VISA 4.1
- NI VXI 3.3.1
- MXI-2 Controller
- 2 Di-050-02 cards and one Di-050-30
- Teradyne Di-Series Driver 2.2.3032.0

Debugging failing tests executed using Algorithmic opcodes may require an extra step when compared to debugging failing tests using conventional pattern memory opcodes. Obtaining the failing pattern index and failing pin does not differ from doing so with conventional memory patterns. However, the failing opcode reported will be the algorithmic opcode from the pattern set. To determine what the failing pin’s expected value was, the pattern index must be mapped to the Deep Serial Memory location for that failure.

Elements of the provided test case can be applied to other large block data transfer applications not explored in this paper. Additional practical applications include boundary scan and loading a large amount of data to an HPROM. Data from ASCII files or other simple sources can easily be programmatically read into a test program and loaded into the channels’ Deep Serial Memory. The framework of basic pattern cycles can be loaded into conventional pattern memory, accessing the Deep Serial Memory for each channel and advancing to the next location. These techniques make the transfer of large blocks of data easier, faster, and more practical.

REFERENCE