

STDF Memory Fail Datalog Standard

A. Khoche¹, J. Katz², S. Landini³, K. Liao⁴, N. Agrawal⁴, G. Plowman⁴, S. Zuo⁴, L. Lai⁵, J. Rowe⁶, T. Zanon⁷
¹Verigy, ²Freescale, ³Arm, ⁴Qualcomm, ⁵Mentor Graphics, ⁶Teradyne, ⁷PDF Solutions

Abstract

Yield learning in modern technologies requires fail data logging from the scan and memory structural tests to gain insight into the failing location inside a chip. Currently there is no standard format to store the fail data in an efficient way. A group of more than 20 companies from ATE, EDA, Semiconductor and Yield Management companies has been working to enhance the Standard Fail Datalog Format (STDF) V4 to enable efficient fail datalog for scan and memory fails. This paper describes the proposed memory fail datalog format.

1 Introduction

Failure analysis and yield learning in modern technologies requires fail data log from volume production to be collected and analyzed by diagnosis tools. Existence of multiple vendors for DFT, ATEs and diagnosis requires data exchange between these vendors. In the absence of a standard format to represent failures, multiple point to point data interchange formats exist today, which lead to unnecessary investment in developing point-to-point translators and potential for error in translations. This paper presents a proposal to standardize the fail data format for memories. The proposed standard extends the existing Standard Test Data Format (STDF) V4, which currently does not support the efficient memory fail datalogging. A group of more than 20 companies from ATE, EDA, Semiconductor and Yield Management companies has been working to create this standard. This paper presents the details of the proposed memory fail datalog standards. The proposed memory fail datalog standard supports both embedded and standalone memories. In case of embedded memories, the standard supports multiple embedded memories with potentially multiple BIST controllers as shown in figure 1. The standard assumes that either BIST controllers communicate the fail data to ATE using a frame structures-based protocol or the output of the memory is directly accessible by ATE. It does not assume any particular implementation of the BIST controller but expects the controller to be able to communicate fail information to the ATE.

2 Overview of STDF

STDF is de facto industry standard and provides a common platform to allow tester, database management systems and data analysis software to store and communicate test data in a general and flexible format. STDF is a binary format where the data is organized in

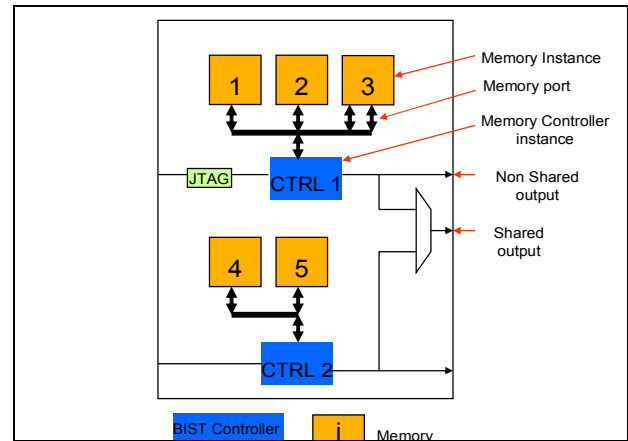


Figure 1: Generic Embedded Memory Structure

the form of a partially ordered set of records. Each record consists of a header and an information section. The header section contains the size of the record, the record type (which indicates the general category of the information) and a record sub type (which indicates the specific information in that category). The information section consists of a sequence of fields of the defined type. STDF defines a set of data-types for these fields. An example of some of the supported data types is shown in table 1.

Table 1: Sample STDF Data Types

Type	Explanation
U*n {n=1,2,4,8}	Unsigned number of n bytes
I*n {n=1,2,4, 8}	Signed Integer of n bytes
C*n	Character array of n bytes
kxT*n	An array of size of T*n type where T is one of the pre-defined types

The record types are organized along the manufacturing entities (e.g., lot, wafer, part, test, test execution). The current version V4 of the standard does not have any predefined records to store scan and memory fails. The standard described in this paper addresses the needs specific to storing memory fail data to volume diagnosis applications in STDF.

3 Fail Datalog Requirements

Figure 2 shows the information objects/classes that need to be supported in the standard to enable an efficient volume diagnosis flow. This object model has been created through the analysis of the design to test to design flow for volume diagnosis. It allows the design information to be carried forward into test environment and the test information to be carried back to analysis tools. The proposed standard maps these object onto various STDF records.

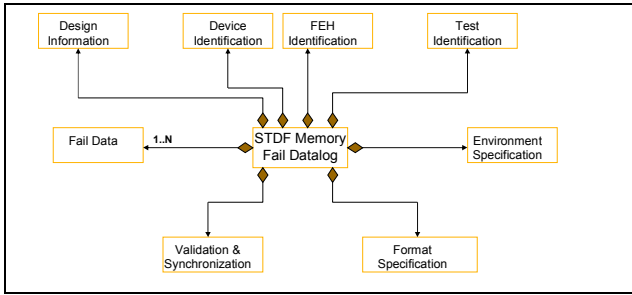


Figure 2: Object Model.

4 STDF Memory Fail Datalog Proposal

The STDF memory fail datalog proposal builds on the STDF V4 format. It uses existing records for the device and equipment identification objects. The records in the existing standard are sufficient for information in these two objects. For rest of the objects new records types have been added to the standard as shown in table 2. The rest of this section describes how the remaining objects are supported using these new records.

Table 2: New Memory Fail Data Records

Type	Sub Type	Record Acronym	Description
1			Data Collection on a per lot basis
	95	ASR	Algorithm Specification Record
	96	FSR	Frame Specification Record
	97	BSR	Bit Stream Specification Record
	99	MSR	Memory Structure Record
	100	MCR	Memory Controller Record
	101	IDR	Instance Description Record
15			Data Collected Per Test Execution
	40	MTR	Memory Test Record

4.1 Design Information

The design information objects contain the information about the design that needs to be passed around the design-to-test-design loop to enable proper diagnosis and data translations. In particular, for the embedded memory architecture example shown in figure 1, the design information object includes information about the controllers, the instances accessed by each controller, the logical memory (e.g. rows/columns, banks), the ports, and orientation of the memory. It also contains a link to the layout file to enable logical to physical mapping for diagnosis and localization of failures. For standalone memories the design information only contains information about the memory model and its orientation. This object is implemented using four STDF records: MSR, MCR, IDR, and MMR (see table 2). The definition of these records is shown in tables 3-6. A single MSR record contains the top level information about the chip/part. A set of Memory Controller Records (MCR) contains the information about the controllers in the designs. Each MCR represents a single controller and contains references to the memory instances it controls. Each memory instance is described using an IDR record

which can be linked to their model information. Figure 3 shows an example of how these records can be used to store design information.

Table 3: Memory Structure Record (MSR)

Field Name	Type	Field Description
NAME	C*n	Name of the design under test
FILE_NAM	C*n	name of the file containing design information
CTRL_CNT	U*2	Count (k) of controllers in the design
CTRLS	kxU*2	Array of controller record indexes

Table 4: Memory Controller Record (MCR)

FieldName	Type	Field Description
CTRL_IDX	U*2	Index of the controller record
NAME	C*n	Name of the controller
INST_CNT	U*2	Count (k) of INST_IDX array
INST_IDX	Kx U*2	Array of memory instance indexes

Table 5: Instance Related Record (IDR)

FieldName	Type	Field Description
INST_IDX	U*2	Unique index of the Instance record
INST_NAM	C*n	Name of the Instance
ORIENT	C*n	Orientation of the instance (two characters)
MIRROR	B*1	Boolean flag to indicate mirrored or not
MODEL_REF	C*n	Pointer to the file describing model (e.g. CTL)
MODEL_ID	U*2	reference to the model record for this instance

Table 6: Memory Model Record (MMR)

FieldName	Type	Field Description
MDL_IDX	U*2	Index of Model record
NAME	C*n	Name of the model
DIM_CNT	U*2	Count (k) of dimensions
DIMS_NAM	kxC*n	Names of of the dimension axis
DIMS_TYPE	kxU*1	Enumeration of types e.g. BANK, COL, COL_MUX etc
DIMS_SIZE	kxU*8	The size of each dimension
DIMS_DIR	kxU*1	0-Not present; 1-Up; 2-Down; 3-Left; 4-Right; 5-CenterOut
PORT_CNT	U*2	Count (k) of Ports in the memory
PORT_TYP	kxC*n	ADDR/DATA/CTRL....
PORT_DIR	C*n	In/Out/InOut
PORT_NAM	kxC*n	Type and name string of the port
PORT_SIZE	kxU*1	The size of port in number of bits
HEIGHT	C*n	Physical height of the memory (with units)
WIDTH	C*n	Physical width of the memory (with units)
FILE_REF	C*n	Name of the file with the physical information for this memory

4.2 Test Information

Test program details are stored in the new Algorithm Sequence Record (ASR). It contains the information about the patterns and algorithms that make up a test in a test program as shown in figure 4. The purpose of this information is to enable

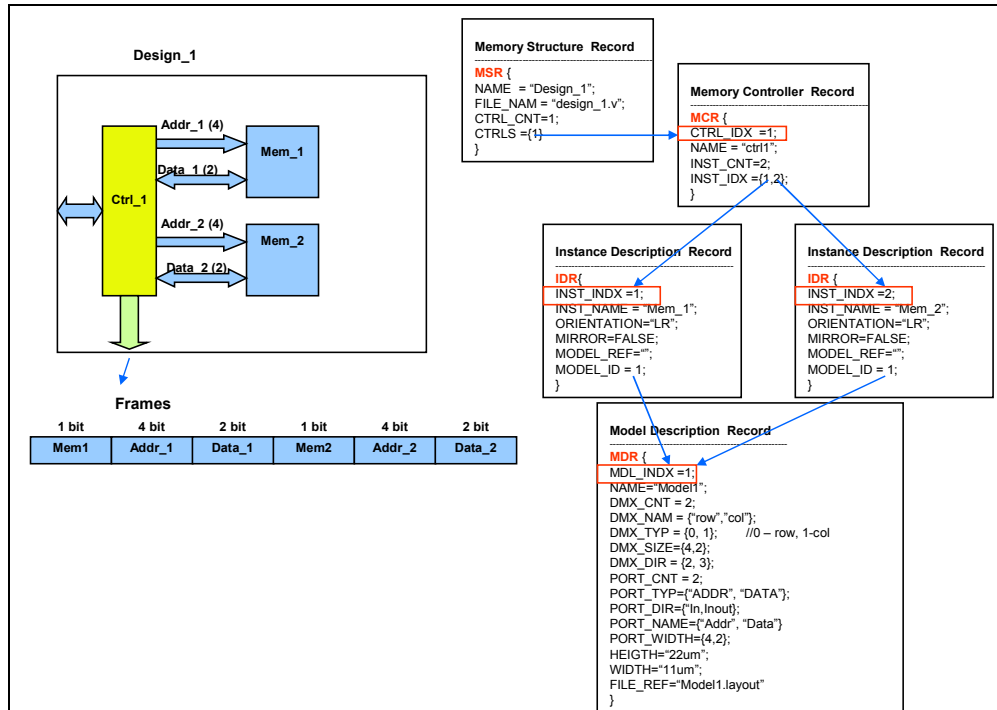


Figure 3: Example Design Description

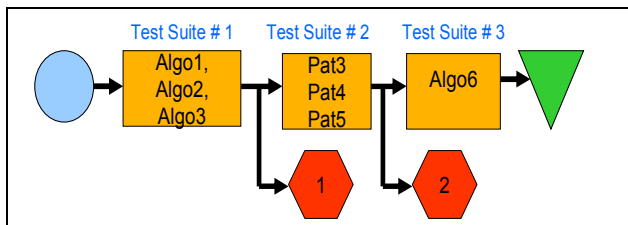


Figure 4: Example Test Flow

translation of cycles to algorithm steps, to rows/cols etc. Field definitions of the ASR are shown in Table 7.. For each test in a test program, this record contains the information on the algorithm/patterns that makeup that test in terms of their name, length, begin and end cycle numbers. There is also provision to specify the name of a file that contains the detailed algorithm description. In addition the test information contains the identification of the test along with its environmental conditions under which the fail log was generated. This information is stored in the MTR record (described later) with the log itself.

Table 7: Algorithm Specification Record

FieldName	Type	Field Description
ASR_INDX	U*2	Unique identifier for this ASR record
STRT_INDX	U*1	Cycle Start index flag
ALGO_CNT	U*1	count (k) of Algorithms descriptions
ALGO_NAM	kxC*n	Name of the Algorithm
ALGO_LEN	kxC*n	Complexity of algorithm, e.g., 13N
FILE_ID	kxC*n	Name of the file with algorithm description
CYC_BGN	kxU*8	Starting cycle
CYC_END	kxU*8	End Cycle number for the algorithm

4.3 Format Specifications Information

The format specification object contains the information on the type and format of the logged fail data. This enables reader to correctly read the fail log. The following format types are supported in the proposed memory fail datalog standard.

4.3.1 (Pin, Cycle no)/(Pin, Address)/(Row, Col) Specification

In this format it is assumed that the outputs of the memory are directly connected to the IO pins, and the failures are directly observed by the ATE which can record the cycle numbers where the fails were observed. It is assumed that the conversion from the (pin, Cycle no) to the (pin, Address) or (Row, Column) will be done by the ATE.

4.3.2 Frame Format Specification

In the case of embedded memories, the fails are communicated to ATE by the BIST controller using frames and a protocol. This format allows the storage of failure frames captured from the BIST controller inside a log. The frame-based format contains two sections: the frame format specification section and the frames themselves. The frame format specification is added in the datalog to enable inter-operability among the custom BIST frame formats, such that a reader could understand the format of the fail log to be able to process the fail frames. Each frame field specification contains four fields: Field Type, Field Name, Field Width, and Default Value. The field types allow the specification of the design entities e.g. controller type, memory identification etc. A frame can have multiple instances of a field type. In addition the frame based format allows a mask

capability to mask certain fields in the portion of a log. Figure 5 shows an example of frame based logging concept

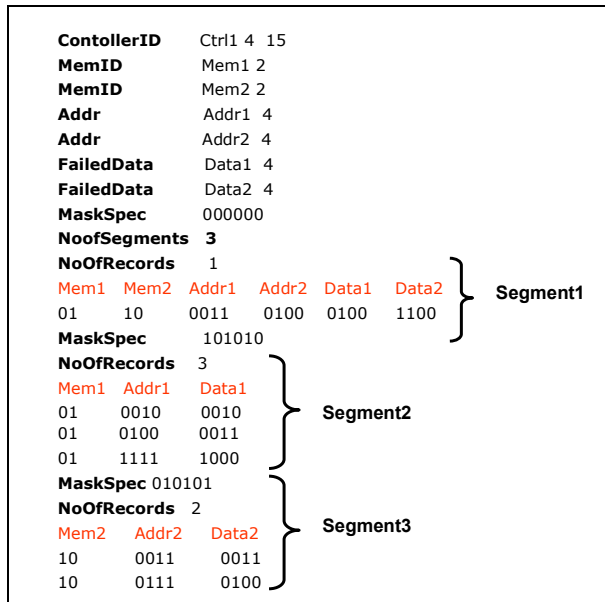


Figure 5: Example Frame Structure

Table 8: Frame Specification Record

FieldName	Type	Field Description
FRM_IDX	U*2	Unique ID for the the frame specification
FLD_CNT	U*2	Count (k) of fields in the frame
FLD_TYP	kxC*n	Field type; one of the supported ones
FLD_NAM	kxC*n	Field Name
FLD_SIZE	kxU*2	field width in number of bits
DEF_VAL	kxU*2	Optional default value

A Frame Specification Record (FSR) shown in table 8 provides a mechanism to define frame formats mentioned above. Each such record has a unique id that is stored in FRM_IDX field. In the actual datalog records a reference to the frame definition is added using the unique id in FRM_IDX field. This enables the reader to identify the frame definition that the frames in the datalog confirm to. The record contains ordered arrays to define each field in the frame. For each field there are four attributes specified as mentioned above these are stored in FLD_TYP, FLD_NAM, FLD_SIZE and DEF_VAL fields. The allowed values for the “Type” field are predefined in the standard. A value has been set for each of the entity in the memory design hierarchy.

4.3.3 Bit stream (Compressed/Uncompressed)

This format allows the raw bit stream to be stored in the log for debug purposes. The data could be uncompressed or compressed. In the case of compressed datalog, the bits are compressed using standard compression algorithm, e.g., zlib. For the uncompressed bit stream, the data volume is reduced by storing only the words that have any fails in them. The log contains the starting address of the words, the count of consecutive words that failed starting from that address and then the word bits as shown in figure 6. Please note that the word bits 0/1

could indicate either the measured 0/1 or it could be used to indicate fails (0-no fail and 1- fail). A flag in the format specification is used to indicate the meaning of bits.

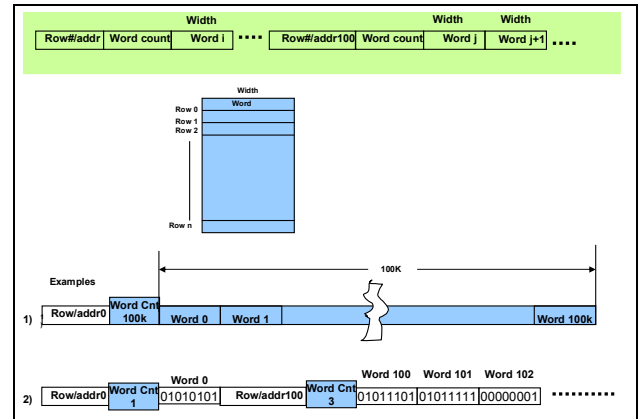


Figure 6: Bit Stream Format

The format for the bit stream is stored in a Bitstream Specification Record (BSR) shown below in table 9. It contains the unique ID (BSD_IDX) for the Bit stream structure definition that is referenced in the actual bit stream fail datalog. The BIT_TYP field indicates the type of the data in the bit stream, The ADDR_SIZE and WC_SIZE fields indicate the size of the address and word count fields in the bit stream. The WRD_SIZE field indicates the width of data words.

Table 9: Bitstream Specification Record

Field Name	Type	Field Description
BSD_IDX	U*2	Unique ID for the Bit stream specification
		Enumerated values to indicate the meanings of the bits in the bit stream where bits are actual data, exp data, fails etc type)
BIT_TYP	U*1	
ADDR_SIZE	U*1	Address field size (U1, U2, U4 or U8)
WC_SIZE	U*1	Word Count Field Size (U1, U2, U4, or U8)
WRD_SIZE	U*2	Number of bits in the word field

4.3.4 Fail Count

The fail count format allows only the fail count instead of actual fails to be stored in the log. This is useful in the volume production for monitoring and statistical analysis. Four levels of abstraction for fail count are supported: Total fail count, Blocks fail count, Row fail count and column fail count. Only Total fail count is mandatory. The fail count format specification is supported in the MTR record that stores the datalog itself. There is no separate record needed to specify this format type.

4.4 Environment, Fail Datalog, Validation Information

The remaining objects in the object model namely Fail datalog, Environment Specification and Validation are mapped to the Memory Test Record (MTR). Table 9 shows the structure of the record and rest of the section describes how these objects are mapped onto this record.

Table 9: Memory Test Record

Field Name	Type	Field Description
CONT_FLG	B*1	Continuation flag
TEST_NUM	U*4	Test number
HEAD_NUM	U*1	Test head number
SITE_NUM	U*1	Test site number
ASR_REF	U*2	ASR Index (Pattern Sequence Record)
COND_CNT	U*2	Count (k) of conditions
COND_LST	kxC*n	Conditions specified as <condition name> = <Value> arrays;
TOTF_CNT	U*8	Total fails during the test
TOTL_CNT	U*8	Total fails logged during test
FILE_INC	B*1	File incomplete
LOG_TYPE	U*1	Type of datalog (Frame, BIT Stream, pin/cycle etc)
ROWF_CNT	U*4	Total count of Failed Rows
COLF_CNT	U*4	Total count of failed columns
BNKF_CNT	U*4	Total count of failed banks
CYC_BASE	U*8	Base cycle count for the cycle based logging
DLOG_MSK	U*1	mask to indicate presence or absence of the field for fail based logging
PIN_CNT	U*4	Count (k) of pins in PIN_ARR
PIN_ARR	kxU*2	Array of PMR indexes for pins
CYC_CNT	U*4	Count k of failed cycles in CYC_OFST ARR
CYC_OFST	kxU*f	array of cycle indexes for each fail
ROW_CNT	U*4	Count (k) of number of rows in the current record
ROW_ARR	k*U*f	Array of row addresses for each fail
COL_CNT	U*4	Count (k) of number of cols in the current record
COL_ARR	k*U*f	Array of column addresses for each fail
STEP_CNT	U*4	Count (k) of march steps stored in this record
STEP_ARR	kxU*1	Array of march steps for each fail
TFRM_CNT	U*4	Total frames in frame based logging
FSEG_CNT	U*2	No of frame segments (i.e. next 4 fields)
FRM_IDX	U*2	Index of the frame record with the frame structure definition
FRM_MASK	B*n	mask for the frame field to indicate presence of absence of a field
FRM_CNT	U*4	Count (k) of frame with current mask
FRAMES	kxU*f	array of frames where the size of frame (f) is derived from the frame definition
BSEG_CNT	U*2	No of bit stream segments
BSD_IDX	U*2	Index of the bit stream record with the definition of bit stream fields
STRT_ADR	U*s	Starting row address in the current segment; s defined in BSR
WORD_CNT	U*w	Count (k) of words in the current stream segment; w defined in BSR
WORDS	kxU*f	The word bits where the word size(f) is defined in the corresponding BSR
BMP_SIZE	U*8	cont (k) of compressed bit map in bytes
CBIT_MAP	kxU*1	Compressed bit map

4.4.1 Environment Conditions

The environment conditions for a test are implemented using a generic <condition=value> concept. It is implemented using COND_CNT and COND_LST field in MTR record.

4.4.2 Validation and Synchronization

A very simple synchronization mechanism has been provided by three fields TOTF_CNT, TOTL_CNT and FILE_INC. The first two stand for total observed failure count and total logged failure count. The total logged fails could be less than total observed fails due to user

limits or hardware limits. The FILE_INC flag indicates the exception condition when the current file can not store all the log information.

4.4.3 Fail Datalog Information

As mentioned in the fail log format section, the standard supports multiple type of datalog. All these types can be stored in this Memory Test Record (MTR). The type of log that is used to store the fail is indicated by the LOG_TYPE field. The valid values for the LOG_TYPE are: (0 – Count only, 1 – Frame Type, 2 – Bit Stream type, 3 – Cycle/(row,col) based logging)

Depending on the type of the datalog the fields that are not part of that type logging are invalid and/or missing.

Fail Count Logging: It is implemented using TOTF_CNT, TOTL_CNT, ROWF_CNT, COLF_CNT and BANKF_CNT as described in table 9.

Frame Based Logging: It is implemented using TFRM_CNT, FSEG_CNT, FRM_IDX, FRM_MASK, FRM_CNT, FRAMES field in MTR. It consists of a series of frame log records. The count of these records is stored in TFRM_CNT field. The frame records are organized in segments. Each segment consists of four fields to describe the frames in the log, the reference to the frame definition (FRM_IDX), the mask specification (FRM_MASK), the number of frames (FRM_CNT) that confirm to the frame definition and the mask and an array of actual frames (FRAMES).

BitStream Based Logging: It is implemented using BSEG_CNT, BSD_IDX, STRT_ADR, WORD_CNT and WORDS fields in MTR. It consists of a series of Bit stream segments. The count of such segments is stored in the BSEG_CNT. For each such segment BS_IDX contains the references to the Bit stream definition record that define the stream structure. The STRT_ADR is the first address where a failure was observed. The WORD_CNT is the count of consecutive words starting from STRT_ADR that also have fails. The WORDS field stores the actual words in array.

Compressed Bit Map Logging It is implemented using following two fields BMP_SIZE and BMP_ARR. BMP_SIZE is the size of compressed bit stream array (BMP_ARR), in bytes, in the current record. If the total bitmap size exceeds the maximum record size (64K) then the remaining data can be stored in the successive MTR records. All such records in this series except the last one have the CONT_FLG set to 1 in the beginning of record to indicate that the data spans over multiple records.

(Pin, Cycle)/(Row, column) Based Logging: A generic structure has been provided to store (Pin, Cycle) and (Row ,Column) based logging. In this structure each information type e.g. row, pin, is stored in its own array. The arrays that are not required can be removed from the log using flag. PIN_ARR field stores the indexes of the pins that failed. The size of this array is determined by the PIN_CNT field.. CYC_OFST array stores the failed

cycle numbers. These number are actually offsets from the based cycle number specified in the CYC_BASE field. This allows 8 byte long cycle numbers without incurring the 8-byte size penalty to each fail. The size of this array is specified in the CYC_CNT field. The ROW_ARR array stores the failed row addresses. Its size is specified in ROW_CNT field. Similarly, COL_ARR stores the failed column addresses. Size is specified in the COL_CNT field. Please note that all these arrays are synchronized by the fails i.e. entry I in the PIN_ARR would correspond to the Ith fail and the CYC_OFST will contain the cycle number for Ith fail as well.

5 Example

An example of frame based logging is show in figure 7 for the embedded memory of figure 3. The example illustrates how the records will be used to store the fails. These examples are shown in ASCII form for readability. The actual records in the log will be binary. Please note that not all the records that will make up the datalog are shown.

6 Conclusion

Yield analysis and improvement in advanced technologies require structural fail information to be collected during volume manufacturing. There is no

standard format to store this information, which leads to a complex environment for collection, storage and processing of this information with a lot of overhead. A new standard format for storing memory fail data presented in this paper has been developed to meet the requirements of volume diagnosis for yield enhancement in advanced technologies with the support from a wide spectrum of companies.

7 References

- [1] Rehnani et al., “ATE Data Collection—A comprehensive requirements proposal to maximize ROI of test”, ITC 2004
- [2] Standard Test Data Format Specification, Version 4.0, Teradyne Inc.
- [3] Constantino, A., “BITMAP Test Data in STDF File”, EMTC 2006.
- [4] Leininger A., et. al., “The Next Step in Volume Scan Diagnosis: Standard Fail Data Format”, ATS 2006.
- [5] A. Khoche, “STDF Fail datalog standardization”: Streamlining the dataflow for volume diagnosis”, EMTC 2007.
- [6] A. Khoche et al, “ A Tutorial on STDF fail data log standardization”, ITC 2008, AIP session 2.

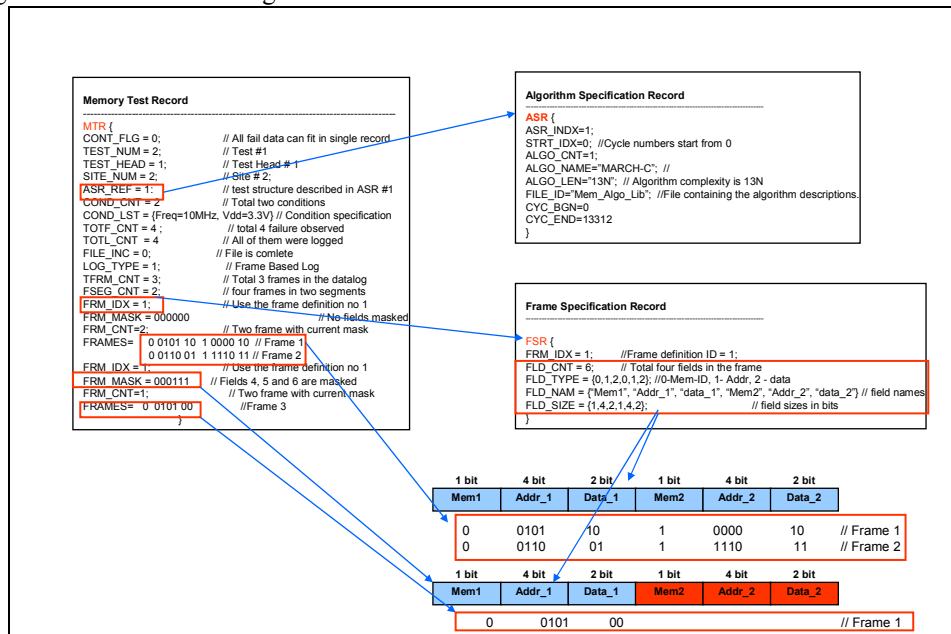


Figure 7: Example of Frame Based Logging