Tabled Data Into Query Able Data, ‘Key’ Concepts Demonstrated Using DDR5 Example

Rahil Jha, Joseph Bauer
Cadence Design Systems (India, USA)
(rahil@cadence.com, jbauer@cadence.com)

Abstract—**Behind every application be it AI, ML, SQL, DBMS, or Verification Intellectual Property (VIP) in the Very Large-Scale Integration (VLSI) domain etc. lots of useful information is encapsulated in tabular form. Extracting and processing the data/information from these tables can be very challenging. Talking about Double Data Rate Generation 5 (DDR5) specification itself has more than 300+ tables which contributes ~20% or more of entire specification and there’s been >50 spec iterations in getting there. The manual extraction of this data and information is prone to error and requires significant efforts.**

Keywords— Very Large-Scale Integration; Functional Verification; Verification Intellectual Property

I. Introduction

Double Data Rate Generation 5 (DDR5) provides memory part bandwidth options via operation over a wide variety of frequencies (3200Mega Transfers per second to 6400+ Mega Transfers per second) [1] and for each of these speeds has a range of sub-part latency option types (low-latency high cost to high-latency low cost: AN, B, BN, C) [1] (and many other part options as well such as densities and stack rank configurations). Many memories parameter characteristic values in DDR5 are linked to the operating speed and sub-part type. Based on a part’s max speed and the current operating speed, these speed dependent parameter values get determined, which in turns helps determine how fast or slow a device can work. Hence the specific level or category of memory speed that a DDR5 memory can operate at is termed speed bin. These parameters are crucial since they relate to various aspects such as command-to-command spacing (tRCD, tRP timings), data access command latencies (Read, Writes, Mode Reg Read) to 1st data burst time (CL), etc. Within the specification these many speed-bins are listed in tabular form with all possibilities of valid and reserved values across speeds and sub-parts across many speed tables. In Cadence Verification IP we devised an ‘Executable Table’ solution to

- Extract valid parameter values correlated in relation with respect to the ‘key’ indexing of tabulated celled data in documented tables,
- Transform this indexable data to executable APIs (and other value items).

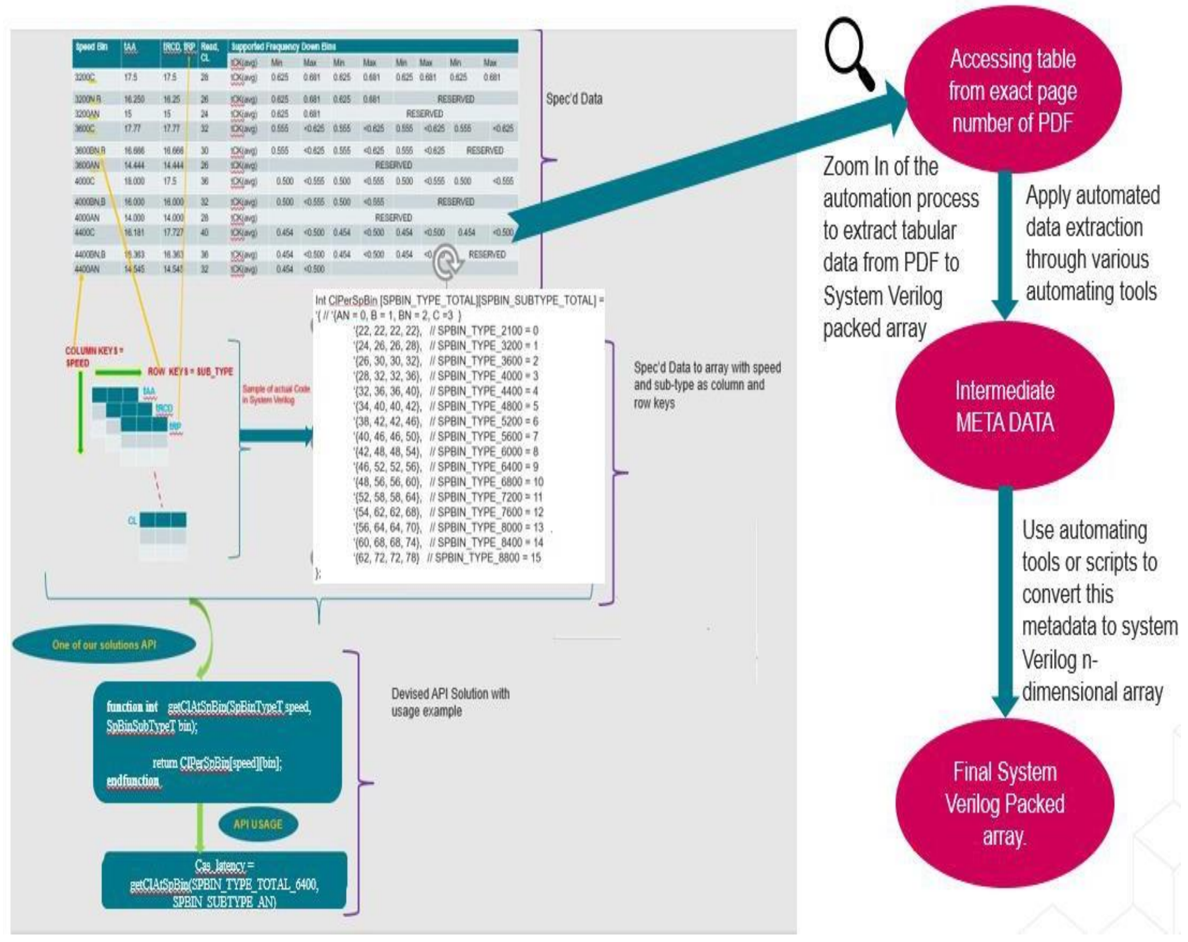
As example the mentioned tabulated data across DDR5’s speed bin tables get extracted and transformed to “Speed to Modes and Timings” Application Programming Interfaces (*abbreviation:* SMT APIs) which can be utilized by DV users of the Verification IP. These SMT APIs provide simulation runtime access to the large collection of speed bin table associated information, selectably returning matching timing and mode parameter value(s) information.

II. Key Innovative Concept

The main concept of this Executable Table we want to convey is the verification technique in making tabled data executable. This includes defining a n-dimensional aggregation matrix/array, where the row ‘key’ is Sub- type and column ‘key’ is Speed in the following DDR5 example (n = 2 in our reduced DDR5 example to convey concept, as additional index dimensions into the tabulated matrix exist such as planar or 3DS selection). Hence the extracted values are a function of index ‘key’ arguments, here: Sub-type x Speed. The API exercises access logic to these dimensional arrays returning the desired values based on the selection ‘key’s, e.g.: Sub-type x Speed. We’ve automated the process in extracting tables from specification pages and converting it to a intermediary metadata. This metadata gets used to achieve our end goal of executable table code generation in any HVL language data structure format (2-D packed array in our example case). There are multiple ways to achieve this automation, although the tabled data extraction details from specification PDF is not covered in this paper. This paper focuses on how the extracted table data is made executable and used to simplify design verification for a robust coverage of specification tables.

III. Executable Table Example - SMT API

Each speed dependent timing parameter has a different value for different operating frequencies and sub-part type. A particular timing parameter value can be valid at a higher operating speed for a given sub-part and be reserved at a lower operating speed for the same sub-part. In such cases, using the SMT API, the VIP user can determine the valid values of timing parameter by passing in the operating speed and the sub-part type. Hence the APIs comes as savior when up-and-down speed bin change scenarios need to be tested. Pictorial representation of the entire problem and solution in Figure 1. Complete Flow from Specification to Implementation:


Speed Bin	tAA	tRCD, tRP	Read, CL	Supported Frequency Down Bins								
				tCK(avg)	Min	Max	Min	Max	Min	Max	Min	Max
3200C	17.5	17.5	28	tCK(avg)	0.625	0.681	0.625	0.681	0.625	0.681	0.625	0.681
3200N,R	16.250	16.25	26	tCK(avg)	0.625	0.681	0.625	0.681	RESERVED			
3200AN	15	15	24	tCK(avg)	0.625	0.681	RESERVED					
3600C	17.77	17.77	32	tCK(avg)	0.555	<0.625	0.555	<0.625	0.555	<0.625	0.555	<0.625
3600BN,R	16.666	16.666	30	tCK(avg)	0.555	<0.625	0.555	<0.625	0.555	<0.625	RESERVED	
3600AN	14.444	14.444	26	tCK(avg)	RESERVED							
4000C	18.000	17.5	36	tCK(avg)	0.500	<0.555	0.500	<0.555	0.500	<0.555	0.500	<0.555
4000BN,B	16.000	16.000	32	tCK(avg)	0.500	<0.555	0.500	<0.555	RESERVED			
4000AN	14.000	14.000	28	tCK(avg)	RESERVED							
4400C	18.181	17.727	40	tCK(avg)	0.454	<0.500	0.454	<0.500	0.454	<0.500	0.454	<0.500
4400BN,B	16.363	16.363	36	tCK(avg)	0.454	<0.500	0.454	<0.500	0.454	[illegible]	RESERVED	
4400AN	14.545	14.545	32	tCK(avg)	0.454	<0.500						

```
Int ClPerSpBin [SPBIN_TYPE_TOTAL][SPBIN_SUBTYPE_TOTAL] =
'{ // '{AN = 0, B = 1, BN = 2, C =3 }
            '{22, 22, 22, 22},  // SPBIN_TYPE_2100 = 0
            '{24, 26, 26, 28},  // SPBIN_TYPE_3200 = 1
            '{26, 30, 30, 32},  // SPBIN_TYPE_3600 = 2
            '{28, 32, 32, 36},  // SPBIN_TYPE_4000 = 3
            '{32, 36, 36, 40},  // SPBIN_TYPE_4400 = 4
            '{34, 40, 40, 42},  // SPBIN_TYPE_4800 = 5
            '{38, 42, 42, 46},  // SPBIN_TYPE_5200 = 6
            '{40, 46, 46, 50},  // SPBIN_TYPE_5600 = 7
            '{42, 48, 48, 54},  // SPBIN_TYPE_6000 = 8
            '{46, 52, 52, 56},  // SPBIN_TYPE_6400 = 9
            '{48, 56, 56, 60},  // SPBIN_TYPE_6800 = 10
            '{52, 58, 58, 64},  // SPBIN_TYPE_7200 = 11
            '{54, 62, 62, 68},  // SPBIN_TYPE_7600 = 12
            '{56, 64, 64, 70},  // SPBIN_TYPE_8000 = 13
            '{60, 68, 68, 74},  // SPBIN_TYPE_8400 = 14
            '{62, 72, 72, 78}  // SPBIN_TYPE_8800 = 15
};
```

```
function int   getClAtSpBin(SpBinTypeT speed,
SpBinSubTypeT bin);

        return ClPerSpBin[speed][bin];
endfunction
```

```
Cas_latency =
getClAtSpBin(SPBIN_TYPE_TOTAL_6400,
SPBIN_SUBTYPE_AN)
```

Spec'd Data
COLUMN KEY$ = SPEED
ROW KEY$ = SUB_TYPE
tAA
tRCD
tRP
CL
Sample of actual Code in System Verilog
Spec'd Data to array with speed and sub-type as column and row keys
One of our solutions API
API USAGE
Devised API Solution with usage example
Accessing table from exact page number of PDF
Zoom In of the automation process to extract tabular data from PDF to System Verilog packed array
Apply automated data extraction through various automating tools
Intermediate META DATA
Use automating tools or scripts to convert this metadata to system Verilog n-dimensional array
Final System Verilog Packed array.

Figure 1 Complete Flow from Specification to Implementation

This Executable Table solution can be seen in above Figure1, from reading the tabular data, eg PDF, and conversion to the aggregation array (SV unpacked array shown) which is the very fundamental building block of this solution.

Again the breakthrough automation recipe is not the scope of this talk, but rather the Executable Table's concept, flow, example, and application overview. In the snippet above, the values of timing parameters CL, tRCD, tRP, tAA are a function of sub-type x speed. The function API's sub-types and speeds arguments are nothing, but the rows and columns of the table given in the data sheet/specification (referred to as 'keys' here). Hence,

$$timing\_parameter\_value = \text{function (Speed x subtype);} \tag{1}$$

Example call below demonstrates the Executable Table usage simplicity via the SMT API :

```
cas_latency = getClAtSpBin(SPBIN_TYPE_3600, SPBIN_SUBTYPE_C);
$display($time,"Printing CL value obtained using API for speed = %s   and sub-
type =%0s: %0d",spbin_type.name , spbin_subtype.name, cas_latency);
```

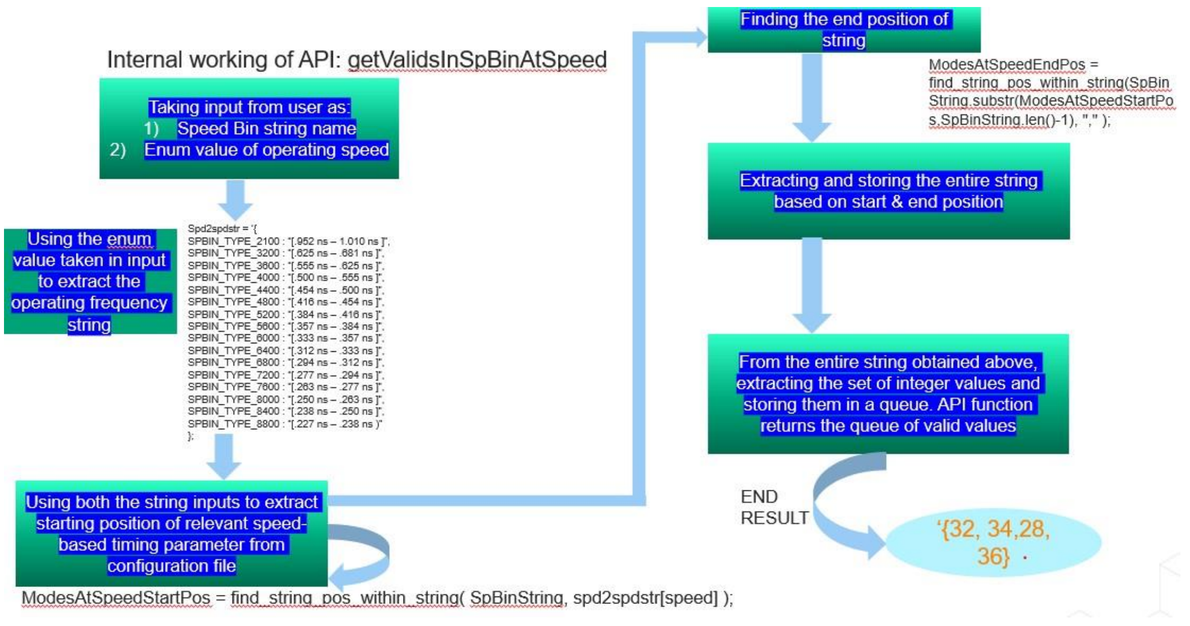
Simulation Output

```
Printing CL value obtained using API for speed = SPBIN_TYPE_3600 and sub-
type=SPBIN_SUBTYPE_C: 32
```

IV. Same Table, Another representative Format

We've also kept in mind the concept of configurability. In real scenarios user might reconfigure some of the values of a speed-bin to different than what is provided in configuration file. Solution must be capable enough to return the updated value.

To address this problem the API's like: getValidsInSpBinAtSpeed takes enum (e.g., SPBIN_TYPE_3200) and timing parameter name string (clSpBin) as an input. The Enum input is then used as an key to find the sub-string of period range (e.g. "[.625ns - .681ns]"). These sub-strings are then used to search for the complete string of valid value sets as per period range from config core string. What gets returned to user is a queue of valid values at a given speed within the full set of valid values across speeds (represented in the part's configuration string attribute, see image with "END RESULT" below).


Internal working of API: getValidsInSpBinAtSpeed
Taking input from user as:
1) Speed Bin string name
2) Enum value of operating speed
Using the enum value taken in input to extract the operating frequency string
Using both the string inputs to extract starting position of relevant speed-based timing parameter from configuration file
Finding the end position of string
Extracting and storing the entire string based on start & end position
From the entire string obtained above, extracting the set of integer values and storing them in a queue. API function returns the queue of valid values
END RESULT
'{32, 34,28, 36} ·

```
Spd2spdstr = '{
SPBIN_TYPE_2100 : "[.952 ns – 1.010 ns ]",
SPBIN_TYPE_3200 : "[.625 ns – .681 ns ]",
SPBIN_TYPE_3600 : "[.555 ns – .625 ns ]",
SPBIN_TYPE_4000 : "[.500 ns – .555 ns ]",
SPBIN_TYPE_4400 : "[.454 ns – .500 ns ]",
SPBIN_TYPE_4800 : "[.416 ns – .454 ns ]",
SPBIN_TYPE_5200 : "[.384 ns – .416 ns ]",
SPBIN_TYPE_5600 : "[.357 ns – .384 ns ]",
SPBIN_TYPE_6000 : "[.333 ns – .357 ns ]",
SPBIN_TYPE_6400 : "[.312 ns – .333 ns ]",
SPBIN_TYPE_6800 : "[.294 ns – .312 ns ]",
SPBIN_TYPE_7200 : "[.277 ns – .294 ns ]",
SPBIN_TYPE_7600 : "[.263 ns – .277 ns ]",
SPBIN_TYPE_8000 : "[.250 ns – .263 ns ]",
SPBIN_TYPE_8400 : "[.238 ns – .250 ns ]",
SPBIN_TYPE_8800 : "[.227 ns – .238 ns )"
};
```

```
ModesAtSpeedStartPos = find_string_pos_within_string( SpBinString, spd2spdstr[speed] );
```

```
ModesAtSpeedEndPos =
find_string_pos_within_string(SpBin
String.substr(ModesAtSpeedStartPo
s,SpBinString.len()-1), "," );
```

Figure 2 Internal Working Of getValidsInSpBinAtSpeed API

In configuration file, each of the speed-based timing parameter gets represented in form of string, as below:

```
clSpBin("[.416 ns - .454 ns) 34 40 42, [.454 ns - .500 ns) 32 36 40, [.500 ns - .555 ns) 32
36, [.555 ns - .625 ns) 26 30 32,  [.625 ns - .681 ns] 24 26 28, [.952 ns - 1.010 ns] 22");
```

A. An example below that demonstrates the user-friendly usage of this API:

Variable declaration in testbench:

```
string  clSpBinString = Sve0.myUvmEnv.activeDevice.config.clSpBin;
valids_t  valCLs ;
SpBinTypeT speed;
```

Testbench API call:

```
speed =  SPBIN_TYPE_4800;
valCLs = getValidsInSpBinAtSpeed( clSpBinString,  speed );
`uvm_info("USER_TEST",$psprintf("\t Printing all possible valid values of CL for 4800 Operating
Frequency %0p",valCLs),UVM_LOW);
```

B. Simulation Output

```
uvm_test_top [USER_TEST]   Printing all possible valid values of CL for 4800 Operating
Frequency '{34,40,42}
```

The result obtained from such API's can be used in different ways. Let's say we are testing randomization scenario where we want to loop over all the valid values of a timing parameter and check the design behavior for that Particular speed, then this API of getValidsInSpBinAtSpeed can be of very useful. There can be many such use cases like this.   This example is given to demonstrate that other useful representative formats can be formed from the same tabulated data set, we've many more such generated executable code from tabled data accounting for user inputs and needed outputs where alternative structure, logic processing gets done.

V. Being Simulation Performance Aware

Reason for choosing 2D unpack array as the basic building block is because they get allotted at build time with fix size which remains same throughout the simulation due to which even if the API get called very frequently during simulation the performance impact will be minimal as compared to when other aggregate data types are used. We architect with associative arrays for speed reasons where direct access key is by name, data is mutable, allocation is sparse, AND not necessary to preserving table data cell order as contiguous ordered sets.. but for ordered things like dates or command sequence order relations these can instead be organized as queue or dynamic array structure. But if data is to be locked, immutable, to data given in document table, then static unpacked array provides optimal compilation optimization for speed and allocation.

VI. Ease Of Use

The feature API is provided directly out-of-box in release product package formatted for user's environment HVL<e.g., SV/UVM/SystemC>, and with example demonstrating ease-of-use of memory configurability and command selection spacing appropriate to operating speeds in user's simulator. No specific configuration is required to enable use of these APIs, even the relevant package import into the DV environment was already done prior for general VIP use, so the VIP user can directly call and use the API methods. Due to such ease of use, this solution was adopted by many customers, internal teams of other groups and our own verification teams.

VII. Realized Effectiveness Of Solution

We saw significant reduction in validation costs (time and effort) versus coverage gain by using SMT APIs [compared to manually transferring speed bin table values from specifications into our internal test scenarios]. This cost reduction and coverage gain is realized over and again with each new revision of table during specification evolution. Simple table, types, and API mechanisms provide configurable access to unpack spec data. Here for a given speed and bin a legal combination {of timings, latencies, modes, command spacings} is returned. But to quantify the scale of improvement consider the task scope in validating a memory model 'scenario of interest' across it's legal intersects of speeds, modes, command spacings, latencies, e.g.:

Scenarios_of_interest[]

...across the legal combinations of...

{speeds [~30], bins [4], modes [~12], affected_Command_to_command_spacings[~20] }..

This blows out to 30x4x12x20 combinations across the scenarios. Here's where the benefit of this devised API solution becomes immeasurable or orders of magnitude better [than the impractical opportunity cost approach of unpacking and hard coding spec table values to a multitude of fixed tests and then MAINTAINING them]. Each test scenario can be checked for compliance across operating speeds applicable to each memory-under-test. Simply iterate speed x bin API inputs to apply appropriate mapped modes, latencies, and timings into DUT stimulus at runtime. Thus, this large noted combinatorial explosion is easily tested with iteration. And maintenance is kept to spec alignment of one speed x bin mapping table of {latency, timing} data.

VIII. Realizing Effectiveness Of Solution On Larger Scale

While DV regression set size scales by 2^n with 'n' test combinations, parallelization of earlier mentioned combinational loop iterations can be realized by fanning out execution of the iteration combinations over large farm of CPU resources:

- Parameterize <speed, bin>scenario block utilizing a macro 'ed version of speed x bin API
- Parallelize simulation runtime speed x bin loop iteration -to- regression test sets[speeds][bins] matrix of test configurations

.. test suite generation = ∑speeds ∑bins [scenario block @ speed=n, bin=m], atomized into separate tests.

Again in same fashion the same extracted tabled data is the source for generating the needed parameterized API's format and regression test set list format for regression.

IX. Actual Customer Testimonial

I recall using Speed to Modes and Timings solution to access valid value of timing parameters based on operating speed and sub-part in the DRAM which allowed us ease of access. Using these valid values, we were able to correctly describe the spacing between valid commands which save our effort from running into command spacing related violations.

X. Application Steps

Executable tables via APIs – how-to considerations in translating spec tables into useful APIs:

1. Identify useful data, i.e. documented table(s) and data subsets in tables.
   Consider which tables, rows, columns, subsections you will be targeting to pull together into an API.
2. Note table's labeling for useful portions of table data, such as table number, subsection, row, columns.
   - Table cells are indexable by table number/row/column/subsection labels.
     How many index dimensions and what are the types for each index used to access down to a cell of atomic table data (consider merging similar repeating tables into one API with additional per Table dimension)
   - Determine the needed API arguments and type, i.e., number of dimensions to index access into this aggregated array/matrix of interesting cell'd data and/or sets (, e.g.). Arguments are often privative types like integers, reals, Enums, strings but can be clubbed together as a struct or class transaction ..)
   - Consider existing DV environment variables and their types, can API use these to access the table data providing internal helpful conversion as needed, e.g.: API can handle argument type

conversion, mapping from existing DV variable types into your API's data structure indexer type (such as period-to-freq, pages-to-bytes, Enums-to-valueID, tuple struct data-to-breakout needed member index privatives).

3. Determine appropriate underlying HVL data aggregation structure for the targeted data such as, e.g. cost/benefit for given API application of
   [static unpacked array -vs- queue -vs- dynamic array -vs- associative array -..]

   Eg:
   - Will Data be immutable, dimensionality shape of aggregation structure fixed and/or static for compiler size allocation and optimization (eg static unpacked array)?
   - Or will data values and shape be subject to change and access at runtime (dynamic arrays, queues, maps aka associative array)
   - Will API always only return single atomic cell of data
   - Or do subsets of data need to be collected through query (eg queue) or directly addressable for speed consideration (map/Assoc-array organizations)?

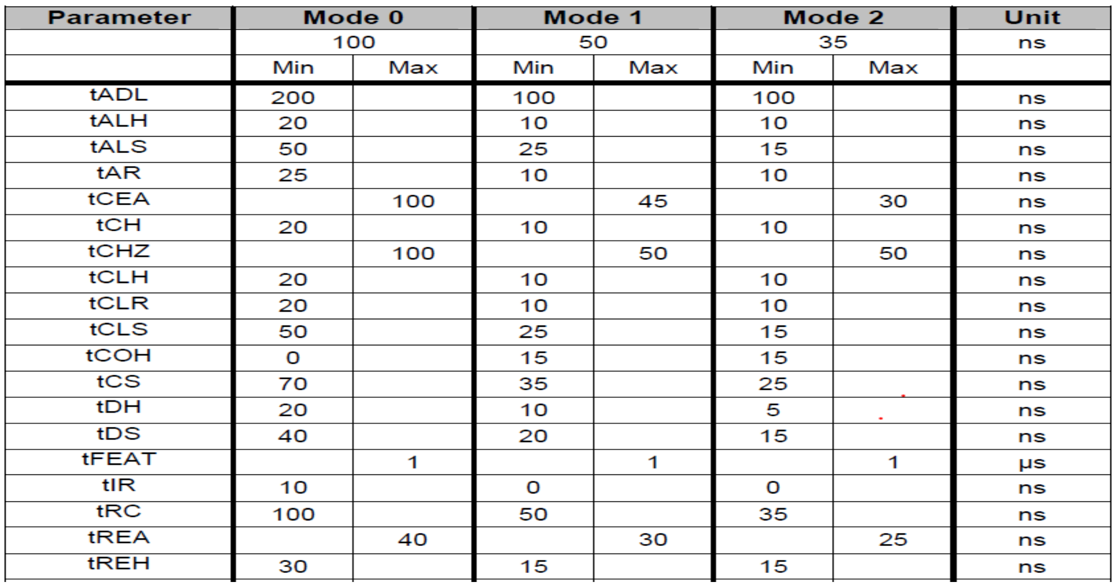
   Does the row/column cell ordering need to be preserved such as for history and sequences (queue, array over maps)

4. Capture API function prototype - Named appropriate to returned table data, with argument types from #2 (needed to index tabled data)
5. Unpack specification table data into the API block with needed HVL format in proper syntax format of chosen data structure, such as in the initialization assignment at data structure declaration.
6. CONSIDER AUTOMATING this #5 format translation if this will be repeated (eg: table value and column/row label pickup and translate with python pandas + regex)
7. Share API for reuse, such as make import'able package or embed into existing already imported package class

XI. Application to Other Protocols

Although this paper demonstrates the ‘Executable Table’ solution with a DDR5 SMT API for speed bin verification, this ‘Executable Table’ solution can also be extended to other protocols. Let’s quickly take an example of ONFI. Referring to v1.0 [2] of the ONFI specification where the minimum and maximum value of a particular timing parameter is based on Mode0, Mode1 or Mode2 as per the “Table 1” below.

Table 1 TIMINGS MODE 0, 1, & 2



Parameter	Mode 0		Mode 1		Mode 2		Unit
	100		50		35		ns
	Min	Max	Min	Max	Min	Max	
tADL	200		100		100		ns
tALH	20		10		10		ns
tALS	50		25		15		ns
tAR	25		10		10		ns
tCEA		100		45		30	ns
tCH	20		10		10		ns
tCHZ		100		50		50	ns
tCLH	20		10		10		ns
tCLR	20		10		10		ns
tCLS	50		25		15		ns
tCOH	0		15		15		ns
tCS	70		35		25		ns
tDH	20		10		5		ns
tDS	40		20		15		ns
tFEAT		1		1		1	µs
tIR	10		0		0		ns
tRC	100		50		35		ns
tREA		40		30		25	ns
tREH	30		15		15		ns

Applying the ‘Executable Table’ API solution, above Table 1 can be represented in form of an 2D packed array whose keys are: Mode, Min-or-Max, so Table’s valid value result for any above-mentioned timing parameters will be:

$$\mathrm{t}DS, \mathrm{t}FEAT, \mathrm{t}ADL, \mathrm{t}CEA \ldots \ldots = function(\,Mode[0,1,2], isMin[T|F]\ \ ) \tag{2}$$

XII. Application Outside Of Engineering Domain

Table 2 Densiy Localities versus Years

YEAR	DENSITY		
	LOCALITY-A	LOCALITY-B	LOCALITY-C
1675	37%	36%	35%
1705	42%	37%	34%
1735	44%	46%	37%
	⋮		
1970	56%	51%	45%
2005	57%	62%	43%

The concept of extracting information from tabular format can be applied across multiple domains and is not limited to the engineering domain. For example, as shown in Table 2 Density Localities versus Years above : let say we’ve data from census carried out over period of years for 3 localities and the census data is formulated in tabular form. Let say we’ve data from census carried out over period of years for 3 localities and the census data is formulated in tabular form. We can apply the same Executable Table concept and steps as the SMT APIs, where the relevant API returns population density as a function of locality x year.

XIII. Conclusion

The Executable Table instruments a specification’s tabled data information in needed DV extractable formats, thus empowering a DV user’s closure of compliance and compatibility over that data without need to refer to the specification. The solution abstracts usage of any spec version as data is KEY sourced from the golden reference table mapping of data, the specification. For ease of compliance coverage, the API’s KEYs align to spec’d table’s rows/columns with argument types whose valid values are both iterate-able and captured in range sets that can be sampled. The Executable Table solution solves access, edit, and exercise of documented table data providing ease-of-use, consistency, readability, efficiency, maintenance, abstraction, and is automatable.

This intentional alignment of coded mapping table to spec table eases maintaining connection to spec so much that the coded translation result, going from spec-to-map over iterations of spec revisions, can be automated. There are many tools for automation which can be used to automate the process of extracting tabled data from specifications and data sheets for conversion to intermediate meta data and from there onwards to needed grammar formats such as HVL mapped table with API logic demonstrated here, thereby automating the entire process of extracting precise data from multiple tabular formatted data.

The benefit of this Executable Table scales with the scope space of a table (layers x rows x columns -> table sets, speeds, bins, modes timings) making the executable table API a necessary building block tool to cover and automate a test scenario’s scope space, accelerating DV’s compliance and compatibility closure with speed, quality, and consistency, ease-of-use. The key concept we demonstrate here is how to limit a DV engineer’s time and effort cost spent in compliance compatibility over tabled data while improving the quality of verification. This executable table solution can be further extended for use in applications which require extraction of data from hefty tabular formats be it in other VIP’s or any software domain or any other domain outside of engineering. Update of values from newer versions of specification into aggregated array data is no more a tedious task and can in fact be automated. Hence, utilizing the very basic functionality of dimensionally aggregated data, we have a maintainable and executable user-friendly solution to ease, accelerate and improve the verification over tabled data. We’ve demonstrated this Executable Table solution with a DDR5SDRAM example and showed it to be generally applicable even outside the scope of DV.

REFERENCES

[1] JEDEC STANDARD DDR5 SDRAM, Revision “JESD79-5A”, October 2021 (Revision of JESD79-5, JULY 2020)
[2] ONFI Specification v1.0: (https://www.onfi.org/specifications)