IEEE Std P1838's flexible parallel port and its specification with Google's protocol buffers

IEEE Std P1838's Flexible Parallel Port and its Specification with Google's Protocol Buffers

Yu Li[2*] Ming Shao[2*] Hailong Jiao[3*] Adam Cron[4] Sandeep Bhatia[5] Erik Jan Marinissen[1,3]

[1] IMEC
Kapeldreef 75
3001 Leuven
Belgium
erik.jan.marinissen@imec.be

[2] KU Leuven – Group T
Andreas Vesaliusstraat 13
3000 Leuven
Belgium

[3] TU Eindhoven
Den Dolech 2
5612AZ Eindhoven
the Netherlands
h.jiao@tue.nl
e.j.marinissen@tue.nl

[4] Synopsys
690 East Middlefield Road
Mountain View, CA 94043
United States of America
a.cron@ieee.org

[5] Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
United States of America
bhatias@google.com

Abstract

IEEE Std P1838 is the DfT standard-under-development for 3D test access into dies meant to be used in 3D multi-die stack assemblies. P1838 is the first DfT standard to include a *flexible parallel port* (FPP): an optional, scalable multi-bit (*'parallel'*) test access mechanism, offering higher test access bandwidth compared to the mandatory one-bit (*'serial'*) port. In this paper, we describe P1838's FPP and propose a formal FPP specification language based on Google's *Protocol Buffers* (PBs), that potentially could become part of the standard. For a realistic example FPP, we provide its formal specification. Finally, we report on a demonstrator software tool, developed by using PBs-generated data access routines, that converts an FPP specification into a corresponding Verilog netlist.

1 Introduction

The market continues to pull for integrated-circuits (ICs) with higher performance, better energy-efficiency, and lower cost. While conventional transistor scaling runs into increasing technical and financial hurdles, the baton in the race to create attractive new IC products that meet market expectations is gradually being taken over by innovations in multi-die stack-assembly and packaging techniques [1, 2]. Large-array fine-pitch micro-bumps implement dense high-performance low-power inter-die interconnects. Through-silicon vias (TSVs) in combination with wafer thinning provide electrical connections via the back-side of a die's substrate, enabling stacks of more than two dies. Packaging costs are drastically reduced by using cheaper materials and wafer-level processes [3]. The I/O-to-pin fan-out functionality of package substrates is replaced by redistribution metal layers, cost-effectively manufactured at wafer level. And the cost of (often expensive) plastic or ceramic packages is circumvented by applying epoxy mold compounds at wafer level. These and other interconnect, assembly, and packaging technology innovations lead to a wide range of multi-die stack architectures, including *Package-on-Package* (PoP), *2.5D-stacked ICs* (SICs) consisting of multiple active dies stacked side-by-side on a passive silicon interposer base [4], *3D-SICs* comprising one or multiple towers of stacked active dies [5, 6], *flip-chip fan-out wafer-level packages* (FC-FOWLP) [7], etc. Both in product variety and volumes we have yet only seen the beginning of multi-die stacks.

Like all micro-electronic products, multi-die stacks need to be tested for manufacturing defects before they can be shipped with acceptable quality levels to their customers. We distinguish the following tests [8]: (1) *pre-bond tests* prior to stacking, (2) *mid-bond tests* on incomplete, partial stacks, (3) *post-bond tests* on complete yet still not packaged stacks, and (4) *final tests* on the final packaged product. The number of possible test flows grows quickly with the number of dies in the stack [9] and hence is the subject of automated trade-off evaluation and optimization [10].

A well-architected on-chip design-for-test (DfT) test access infrastructure is indispensable for achieving a high-quality test. Conventional ('2D') DfT structures such as internal scan chains, test data compression, IEEE Std 1500 wrappers around embedded cores, and/or built-in self-test (BIST) engines are required to provide test access within a single die. In particular for test operations *after* stack assembly has commenced (i.e., mid-bond, post-bond, and final tests), we also need novel '3D' DfT structures that provide modular [11] test access from (and to) the external stack I/Os to (and from) the various dies and inter-die interconnects that require testing. In a typical die stack, the external stack I/Os connect to the bottom side of the base die. In that case, test stimuli need to be transported through other dies in the stack up to the die where they are meant to execute their defect detection work; likewise, test responses need to be transported from the die-under-test through other dies in the stack down to the external stack I/Os. Several ad-hoc 3D-DfT architectures have been proposed [12–18]; these architectures do their job, but are not fully compatible with each other. However, it is important to guarantee interoperability of the 3D-DfT architecture across the various dies in a stack, even if these dies are designed by different teams or companies; this requires standardization.

IEEE Std P1838 is a standard (currently still under development) for 3D-DfT [19, 20]. It standardizes per-die 3D-DfT features, such that if compliant dies are brought together in a die stack, a basic minimum of test access is guaranteed to work across the stack. IEEE Std P1838 consists of three main components: (1) a *serial control mechanism* (SCM), (2) a *die wrapper register* (DWR), and (3) a *flexible parallel port* (FPP). SCM and DWR are 3D extensions of existing standards, respectively IEEE Std 1149.1 [21] and IEEE Std 1500 [22]. P1838's FPP is an optional, scalable multi-bit test access mechanism that offers higher bandwidth compared to the one-bit ('serial') mandatory part of P1838. Standardizing multi-bit test access is a novelty: IEEE Std 1149.1 is fully based on serial ac-

[*] Yu Li is currently with the Dept. of Computer Science and Engineering of the Chinese University of Hong Kong, yuli@cse.cuhk.edu.hk; Ming Shao is currently with Punch Powertrain, ming.shao@punchpowertrain.com; Hailong Jiao is currently also with the Shenzhen Graduate School of Peking University, jiaohl@pkusz.edu.cn.

978-1-5386-3728-9/18/$31.00 ©2018 IEEE

cess and although IEEE Std 1500 mentions a wrapper parallel port (WPP), it leaves its definition entirely open. In that light, P1838's FPP is a novel structure, natively developed by the P1838 Working Group. P1838's FPP, while still under development, was first published in [20]. Since that publication, the FPP definition has undergone several impactful changes that introduce the notion of *paths* and assign symmetrical roles to the various FPP terminals. *This* paper describes the FPP and proposes an FPP specification and description language based on Google's *Protocol Buffers*. The choice for PBs brings platform- and programming-language-independent compiler support, and offers backward compatibility in case of future extensions of the FPP specification language.

The rest of this paper is organized as follows. Section 2 gives a brief overview of IEEE Std P1838, while Section 3 details its FPP. Section 4 introduces PBs. In Section 5, we propose an FPP specification language based on PBs. We present an illustrative, realistic FPP example in Section 6, for which we provide the formal FPP specification as well as the automatically generated gate-level netlist. Section 7 concludes this paper.

2 Overview of IEEE Std P1838

The aim of IEEE Std P1838 is to define at die-level, in addition to die-internal ('2D') digital DfT features which are not governed by P1838, standardized and scalable 3D-DfT features, such that when compliant dies are stacked, a stack-level 3D-DfT test access architecture emerges with guaranteed minimum functionality and many optional extensions [20]. The P1838 test access architecture supports *modular* testing [11], in which dies and interconnect layers between adjacent stacked dies can be tested individually.

To support the many possible stack architectures, P1838 avoids referring to relative physical positions of adjacent dies ('top', 'bottom'). Instead, the standard-under-development considers the order in which the various stacked dies connect to one another and to the external stack I/Os, through which external test equipment needs to apply stimuli to and capture responses from all dies and interconnect layers. P1838 assumes that all external I/Os of a stack are concentrated in a single die. Each die has exactly one *primary* interface: the collection of signals which connect to the *previous* die, i.e., the die in the direction of the external stack I/Os; apart from the *first* die, for which its primary interface *is* the external stack interface. The collection of signals going to a *next* die is referred to as a *secondary* interface of this die. The primary interface of the next die plugs into the secondary interface of this die. A die can have zero or more secondary interfaces. A middle die in a single-tower stack has exactly one secondary interface. The *last* die in a stack tower has no secondary interface. And dies which connect to multiple next dies (a.k.a *multi-tower* architectures [23, 24]) have multiple secondary interfaces, one for each stacked die.

P1838's 3D-DfT consists of three components: (1) SCM, (2) DWR, and (3) FPP. The main purpose of the SCM is to configure the dies in the stack into one of their test modes. In addition, the SCM provides low-bandwidth test data access at one bit per clock cycle. The primary interface of a compliant die is extended with an IEEE Std 1149.1-compliant five-terminal test access port (*primary TAP*) and associated 16-state TAP Controller finite state machine and decode logic [21]. The TAP provides a serial test access mechanism via its test-data-in (TDI) and test-data-out (TDO) terminals. Each secondary interface is extended with a *secondary TAP* with the same five signals as a primary TAP, but with reversed direction. The SCM also includes a *TAP configuration register* (TAP-CR), hooked up to the TAP Controller as a dedicated test data register. Through the TAP-CR, the user can activate individual secondary TAPs, such that the SCM of the corresponding next die is included in the stack's serial test access path.

The DWR mandates a *wrapper cell* at the vast majority of digital signals of the primary and secondary interfaces of a die. The wrapper cells provide scan test access and hence controllability and observability. Wrapper technology enables modular testing [11], in which large stack designs are divided into smaller units, such that ATPG is performed on tractable units, the responsability for test coverage is assigned to whom it belongs (the die maker), test data volume is reduced [25], and reuse in subsequent stack designs becomes easier. Dies can be tested internally while the wrapper is in its inward-facing mode (INTEST) and provides isolation from die-external switching. We can also test the interconnects between adjacent dies with the wrapper in its outward-facing mode (EXTEST). P1838's wrapper cell definition is largely based on the wrapper boundary register (WBR) definition of IEEE Std 1500 [22]. Like IEEE 1500, P1838 is quite liberal with respect to wrapper cells; multiple shift bits, an update register, and output guarding are all optional. P1838 allows wrapper cells to be either *dedicated* or *shared*, i.e., implemented by reusing already-present functional flip-flops. DWRs are mostly thought to exist at the die boundary, but P1838 also allows *'inland'* wrapper cells with some combinational ('shore') logic between DWR and actual die boundary, to facilitate flip-flop sharing and at-speed interconnect testing [20, 26].

Configured via the SCM, P1838 supports many configurations [27], which are characterized by (1) which test ports are utilized and (2) which DfT elements are included into the active test access between these ports. We distinguish *serial* vs. *parallel* modes, in which there is either a one-bit test access path between TDI and TDO, or a multi-bit test access path between the FPP terminals. And we distinguish INTEST vs. EXTEST, in which either the DWR cells alone, or in combination with 2D-DfT scan paths are selected between the active test access terminals.

3 P1838's Flexible Parallel Port

The main purpose of P1838's FPP is to provide more test bandwidth than the one-bit test access via the mandatory TAP. As there is no 'one-size-fits-all' requirement for test access bandwidth, the FPP is optional and scalable in many parameters. It provides a flexible template that covers common, advanced, and even exotic test scenarios.

A key element in the definition of the FPP is a *lane*, i.e., a one-bit test data transportation hub. The lane template has six terminals.

- FPP_PRI and FPP_SEC (bidirectional): for vertical transport to and/or from the previous and next die in the die stack, respectively. Vertical inter-die interconnects as micro-bump and/or TSVs take up a relatively large circuit area. Therefore, vertical input and output interconnects often get combined into bidirectional terminals, which improves efficiency

w.r.t. the available interconnect resources.

- FPP_TO_SIDE (output) and FPP_FROM_SIDE (input): for transport to/from another lane in the same die. The FPP_TO_SIDE output of one lane connects to the FPP_FROM_SIDE input of another lane.
- FPP_TO_CORE (output) and FPP_FROM_CORE (input): for transport to/from the core-under-test in the same die.

A lane instantiation implements one or more *paths* that uses these terminals as input *source* or output *destination*. The maximum number of possible paths in one lane is $(4 \times 4) - 2 = 14$ (as the bidirectional terminals FPP_PRI and FPP_SEC cannot serve simultaneously as source *and* destination of the same path). Typical cases implement only a few paths per lane. If multiple paths share a common destination, a multiplexer is required to perform selection.

P1838 distinguishes registered and non-registered lanes. *Registered lanes* transport data that can be pipelined, such as scan test data. Registered lanes require a clock input, named FPP_CLK_IN. Paths in registered lanes can be provisioned with pipeline registers to increase throughput. It is allowed to share (some of) these pipeline registers between different paths, if the die layout so permits. Optionally, a bypass for pipeline registers can be provided. The destination terminals in a registered lane are equipped with a *hold element* (e.g., a latch at the inverted clock, a.k.a. 'lock-up' latch) to prevent timing issues in the test data transport, since especially the inter-die interconnect is sensitive to hold-time violations and races. *Non-registered lanes* are used for transporting signals that cannot be pipelined (such as clock signals); these lanes obviously do not contain pipeline registers nor hold elements. A non-registered lane serving as clock source has an FPP_CLK_OUT output, which is meant to be connected to the FPP_CLK_IN inputs of the registered lanes that use this clock signal.

A lane is controlled by configuration bits that enable the output drivers of the bidirectional terminals FPP_PRI and FPP_SEC that connect to off-chip circuitry, multiplexer selection signals, and pipeline bypass signals, if present.

Figure 1 shows a schematic example of a registered lane. The data terminals in Figures 1, 4, and 5 have, for illustration purposes, been assigned a color code: dark and light green for resp. FPP_PRI and FPP_SEC, dark and light blue for resp. FPP_FROM_SIDE and FPP_TO_SIDE, and red and orange for resp. FPP_FROM_CORE and FPP_TO_CORE. The clock signal is shown in purple and the configuration controls are black. This example lane does not use the terminal FPP_FROM_SIDE and hence this terminal is grayed out in Figure 1. The example lane uses the bidirectional terminals FPP_PRI and FPP_SEC only as input and output, respectively; their superfluous drivers and the output enable signal FPP_PRI_EN are also grayed out in the figure. This lane has five paths: (1) FPP_PRI $\rightarrow$ FPP_SEC, (2) FPP_PRI $\rightarrow$ FPP_TO_SIDE, (3) FPP_PRI $\rightarrow$ FPP_TO_CORE, (4) FPP_FROM_CORE $\rightarrow$ FPP_SEC, and (5) FPP_FROM_CORE $\rightarrow$ FPP_TO_SIDE. Destinations FPP_SEC and FPP_TO_SIDE both serve two paths, and therefore are both equipped with a selection multiplexer, controlled by the inverted version of configuration register bit BYPASS. Paths (1) and (2) both contain a single pipeline register without bypass; the other three paths have no pipelines. All three destinations have a lock-up latch as mandatory hold element.

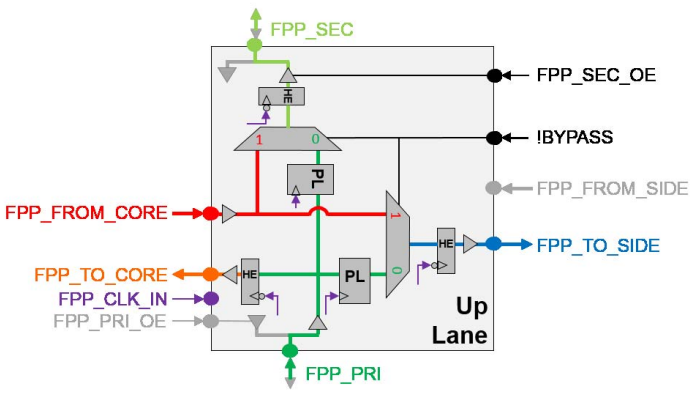

Figure 1: Example FPP lane comprising five paths.

The example in Figure 1 corresponds to the upward parallel test access mechanism (PTAM) in a 3D-SIC in [12–15, 27], described in the context of P1838's FPP as *conventional parallel port* (CPP) in [20]. The only (minor) difference with this prior PTAM implementation is the addition of a lock-up latch on the FPP_TO_CORE output terminal, to make also that output robust for clock-skew.

A *channel* is a set of identical lanes, controlled by the same configuration bits, and, in case the lane is registered, serviced by the same clock lane. An FPP consists of one or more channels. It is permitted that multiple channels share configuration bits and/or clock signals.

4 Google's Protocol Buffers

Protocol Buffers (PBs) is "a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler" [28]. Google has developed PBs as internal *lingua franca* for use in communication protocols and data storage in the context of its distributed data centers [29]. In 2008, Google made PBs available to the public under an open-source license.

In PBs, data is structured in *messages*. A message holds one or more fields. Each field consists of a name-value tuple. *Names* are user-defined key words, while *value* types can be integer or floating-point numbers, booleans, strings, raw bytes, or other PBs message types, allowing hierachically-structured data. Fields are specified to be *required* ($= 1\times$), *optional* ($0\times$ or $1\times$), or *repeated* ($\geq 0\times$).

Each field is assigned a *tag number* unique within its message. This allows to extend an existing PBs data structure with new fields, without breaking backwards-compatibility of existing code. Existing fields keep their existing tag number, while new fields get assigned a fresh, so-far unused tag number, unique within the message. Application programs based on older versions of the data structure definition without these new extensions will simply ignore the new fields when parsing data.

Messages are defined in a `.proto` text file. That `.proto` file serves as input for Google's PBs Compiler, which generates source code for data access classes, containing routines specific to the data format, as well as methods to parse the entire data struc-

ture to/from ASCII or binary data streams. The PBs Compiler is platform-neutral, i.e., it runs on Linux, MacOS, and Windows. The PBs Compiler is also language-neutral: version 2 (*proto2*) generates source code in the programming languages C++, Java, and Python, while version 3 (*proto3*) in addition supports C#, Java Lite, JavaScript, Objective-C, and Ruby [28].

Practical usage of PBs is as follows. A (possibly hierarchical) structured data format is defined through a `.proto` file. On a compute platform of choice, the PBs Compiler generates numerous test data access routines in a programming language of choice. These routines can be used by a programmer to implement application software that can read, write, and manipulate data in the defined data structure, either in human-readable text format or in more-compact binary format. Based on the same `.proto` source file, more applications can be written, possibly on another platform and/or in another programming language, all capable of handling and exchanging the same data files.

In the context of this paper, we used PBs to define an FPP specification language and demonstrate its usage by developing example application software for it. The corresponding software flow is shown in Figure 2. We first developed an example FPP specification in a self-defined `.fpp` file format, respecting the constraints and limitations imposed by PBs. Subsequently, in a straightforward, almost mechanic manner, we derived the corresponding FPP language definition in the `fpp.proto` file.[1] `fpp.proto` then served as input source file for the PBs Compiler to generate data access routines on a platform and in a programming language of choice; we chose Linux and C++. With these automatically generated routines, we built as demonstrator application a software tool that reads an FPP specification in the `.fpp` format and automatically generates a corresponding Verilog gate-level netlist [30].

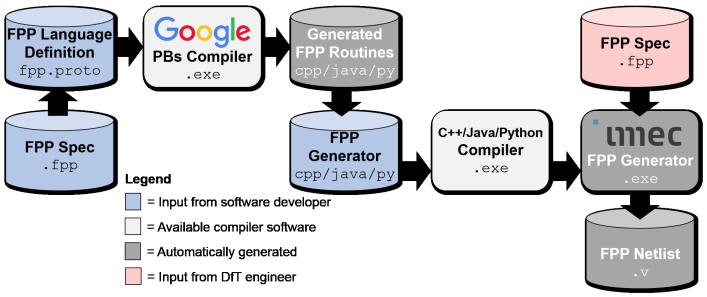

Figure 2: Flow for the definition of an FPP specification language and development of an FPP spec-to-netlist generator as demonstrator application.

5 FPP Specification Language Definition

Figure 3 shows the definition of our FPP specification language in PBs proto2. An FPP has a *name* and consists of an optional configuration register, and a number of non-registered lanes and channels (Lines 03–06). [1]

The FPP specification language has its data structured in messages that consist of fields which are either *strings* (Lines 03, 09, 16, 19, 48, 62, 71, and 74), *integers* (Lines 12, 49, and 73), dedicated enumerated types (Lines 40–46, 77–82, and 83–88), or other messages. Message fields have tag numbers starting with '=1', while enumerated data types have tag numbers starting with '=0'.

```
01: syntax = "proto2";
02: message FppDef {
03:   required string name                =1;
04:   optional ConfigRegDef configReg     =2;
05:   repeated NonRegLaneDef nonRegLane   =3;
06:   repeated ChannelDef channel         =4;
07: }
08: message NetDef {
09:   repeated string net                 =1;
10: }
11: message ConfigRegDef {
12:   required int32 bitCount             =1;
13:   repeated ConfigBitDef configBit     =2;
14: }
15: message ConfigBitDef {
16:   required string bit                 =1;
17: }
18: message NonRegLaneDef {
19:   required string name                =1;
20:   repeated NonRegDestDef destination  =2;
21:   optional NetDef pri                 =3;
22:   optional NetDef sec                 =4;
23:   optional NetDef toSide              =5;
24:   optional NetDef fromSide            =6;
25:   optional NetDef toCore              =7;
26:   optional NetDef fromCore            =8;
27:   optional NetDef clkOut              =9;
28:   optional ConfigBitDef priOutEn      =10;
29:   optional ConfigBitDef secOutEn      =11;
30: }
31: message NonRegDestDef {
32:   repeated ConfigBitDef muxCtrl       =1;
33:   required NonRegDestList port        =2;
34:   repeated NonRegPathDef path         =3;
35: }
36: message NonRegPathDef {
37:   optional string muxCtrlVal          =1;
38:   required SourceList source          =2;
39: }
40: enum NonRegDestList {
41:   FPP_PRI                             =0;
42:   FPP_SEC                             =1;
43:   FPP_TO_SIDE                         =2;
44:   FPP_TO_CORE                         =4;
45:   FPP_CLK_OUT                         =6;
46: }
47: message ChannelDef {
48:   required string name                =1;
49:   required int32 regLaneCount         =2;
50:   required RegLaneDef regLane         =3;
51:   optional NetDef clkIn               =4;
52:   optional NetDef pri                 =5;
53:   optional NetDef sec                 =6;
54:   optional NetDef toSide              =7;
55:   optional NetDef fromSide            =8;
56:   optional NetDef toCore              =9;
57:   optional NetDef fromCore            =10;
58:   optional ConfigBitDef priOutEn      =11;
59:   optional ConfigBitDef secOutEn      =12;
60: }
61: message RegLaneDef {
62:   required string name                =1;
63:   repeated RegDestDef destination     =2;
64: }
65: message RegDestDef {
66:   repeated ConfigBitDef muxCtrl       =1;
67:   required RegDestList port           =2;
68:   repeated RegPathDef path            =3;
69: }
70: message RegPathDef {
71:   optional string muxCtrlVal          =1;
72:   required SourceList source          =2;
73:   optional int32 plRegs               =3;
74:   optional string plTriggerEdges      =4;
75:   optional ConfigBitDef plBypass      =5;
76: }
77: enum SourceList {
78:   FPP_PRI                             =0;
79:   FPP_SEC                             =1;
80:   FPP_FROM_SIDE                       =3;
81:   FPP_FROM_CORE                       =5;
82: }
83: enum RegDestList {
84:   FPP_PRI                             =0;
85:   FPP_SEC                             =1;
86:   FPP_TO_SIDE                         =2;
87:   FPP_TO_CORE                         =4;
88: }
```

Figure 3: Definition of the FPP specification language in PBs *proto2* format.

There is a strong need to be able to express arrays in the FPP specification language. Unfortunately, PBs does not support arrays natively. We have circumvented this issue by expressing array constructs in *string* types. Arrays of interconnect *nets* (a.k.a. *buses*) can be expressed as a string that contains the net name, followed by an array range (Line 08-10); e.g., "bus[7:0]". We also allow forking, merging, and reversing of buses, e.g., "bus1[7:4] + bus2[0:3]". In Line 74, we specify a string *plTriggerEdges* to express the polarity of the trigger edges (positive or negative) of the pipeline registers in a path as a sequence of "P" and "N" characters, one for each subsequent pipeline stage.

Application software can implement the following semantic checks for parsing an `.fpp` file.

- A (non-registered or registered) lane has at least one *destination* (Lines 20 and 63).
- Every destination of a (non-registered or registered) lane has a number of paths between 1 and s, where s is the size of the enumerated type *SourceList* (Line 77). FPP_PRI and FPP_SEC can be both source and destination, but not simul-

[1] If our FPP specification language `.fpp` were to be adopted as standard, it is the `fpp.proto` file that should be released as standardized defintion of the language.

taneously for the same path.

- For each destination d of a (non-registered or registered) lane, the number of multiplexer select bits *muxCtrl* (Lines 32 and 66) shall at least be $\lceil \log_2(p) \rceil$, where p is the number of *paths* (Lines 34 and 68) corresponding to destination d.
- The string *muxCtrlValue* (Lines 37 and 71) shall consist exclusively of "0" and "1" characters and shall in length be equal to the number of configuration bits *muxCtrl* (Lines 32 and 66) of the destination in question.
- In *ConfigRegDef*, the number of *ConfigBits* (Line 13) shall equal *bitCount* (Line 12).
- The *muxCtrl* bits used in *NonRegDestDef* (Line 37) and *RegDestDef* (Line 66) shall be defined as *configBits* in *ConfigRegDef* (Line 13).
- The nets *pri*, *sec*, *toSide*, *fromSide*, *toCore*, and *fromCore* (Lines 52–57) of a *ChannelDef* shall describe arrays of nets, each with width *regLaneCount* (Line 49).
- The integer *plRegs* (Line 73) shall be non-negative.
- The string *plTriggerEdges* (Line 74) shall consist exclusively of "P" and "N" characters and shall in length be equal to *plRegs* (Line 73).

6 FPP Specification Language Example

In this section, we provide a realistic, illustrative example of an FPP specification and application software that utilizes that specification. For the example FPP, we have selected the *Test@First* architecture, in which PTAMs on their way up into the stack can be configured to either (1) test the die's core circuitry or (2) bypass it, and subsequently either (1) continue to travel upward into the die stack via its secondary interface to the next die, or (2) turn around down via the primary interface to the stack's external I/Os (thereby possibly passing previous dies) [12–15, 20, 27].[2] The FPP example is depicted in Figure 4. It contains three types of lanes: a channel with eight 'up' lanes, a channel with eight 'down' lanes, and a non-registered lane that serves as clock lane to the two abovementioned channels. Note that the 'up' lanes are identical to the registered lane featured in Figure 1. The configuration signals come from a Configuration Register that in turn can be controlled as test data register from the TAP [21].

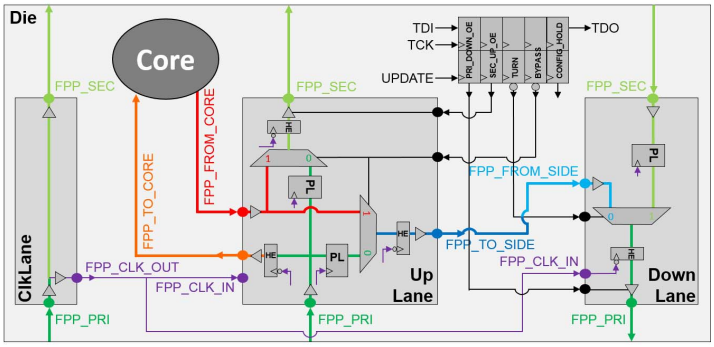

Figure 4: FPP example following the Test@First FPP architecture.

Figure 5 gives the full specification of the *Test@First* FPP architecture in the PBs `.fpp` format defined in the previous section. Note that Figures 1, 4, and 5 use the same color coding for the FPP terminals. The full FPP specification in `taf.fpp` for this realistic FPP consists of only 94 lines.

In this FPP example, the bidirectional terminals FPP_PRI and FPP_SEC are used unidirectionally only. Consequently, the output enable signals SEC_UP_OE and PRI_DOWN_OE can be always 'on'. Modification of `taf.fpp` in Lines 068 and 093 to respectively

068: **secOutEn** {bit: "1"}
093: **priOutEn** {bit: "1"}

```
001: name: "FPP_TEST_AT_FIRST"

002: configReg {
003:     bitCount: 4
004:     configBit {bit: "BYPASS"}
005:     configBit {bit: "TURN"}
006:     configBit {bit: "SEC_UP_OE"}
007:     configBit {bit: "PRI_DOWN_OE"}
008: } //end configReg

009: nonRegLane {
010:     name: "ClkLane"
011:     destination {
012:         port: FPP_CLK_OUT
013:         path {source: FPP_PRI}
014:     }
015:     destination {
016:         port: FPP_SEC
019:         path {source: FPP_PRI}
018:     }
019:     pri {net: "TestClock"}
020:     sec {net: "TestClock_UP"}
021:     clkOut {net: "FppClock"}
022:     secOutEn {bit: "SEC_UP_OE"}
023: } // end nonRegLane ClkLane

024: channel {
025:     name: "UpChannel"
026:     regLaneCount: 8

027:     regLane {
028:         name: "UpLane"

029:         destination {
030:             port: FPP_TO_CORE
031:             path {source: FPP_PRI}
032:         }

033:         destination {
034:             muxCtrl: "!BYPASS"
035:             port FPP_SEC
035:             path {
037:                 muxCtrlVal: "0"
038:                 source: FPP_PRI
039:                 plRegs: 1
040:                 plTriggerEdges: "P"
041:             }
042:             path {
043:                 muxCtrlVal: "1"
044:                 source: FPP_FROM_CORE
045:             }
046:         }

047:         destination {
048:             muxCtrl: "!BYPASS"
049:             port: FPP_TO_SIDE
050:             path {
051:                 muxCtrlVal: "0"
052:                 source: FPP_PRI
053:                 plRegs: 1
054:                 plTriggerEdges: "P"
055:             }
056:             path {
057:                 muxCtrlVal: "1"
058:                 source: FPP_FROM_CORE
059:             }
060:         }
061:     }

062:     clkIn {net: "FppClock"}
063:     pri {net: "PRI_UP[7:0]"}
064:     sec {net: "SEC_UP[7:0]"}
065:     toSide {net: "LaneConn[7:0]"}
066:     toCore {net: "TO_CORE[7:0]"}
067:     fromCore {net: "FROM_CORE[7:0]"}
068:     secOutEn {bit: "SEC_UP_OE"}

069: } // end channel UpChannel

070: channel {
071:     name: "DownChannel"
072:     regLaneCount: 8
073:     regLane {
074:         name: "DownLane"
075:         destination {
076:             muxCtrl: "!TURN"
077:             port: FPP_PRI
078:             path {
079:                 muxCtrlVal: "0"
080:                 source: FPP_FROM_SIDE
081:             }
082:             path {
083:                 muxCtrlVal: "1"
084:                 source: FPP_SEC
085:                 plRegs: 1
086:                 plTriggerEdges: "P"
087:         } }
088:     }
089:     clkIn {net: "FppClock"}
090:     pri {net: "PRI_DOWN[7:0]"}
091:     sec {net: "SEC_DOWN[7:0]"}
092:     fromSide {net: "LaneConn[7:0]"}
093:     priOutEn {bit: "PRI_DOWN_OE"}
094: } // end channel DownChannel
```

Figure 5: Specification `taf.fpp` of the Test@First architecture as P1838 FPP.

[2] In the similar *Test@Last* architecture, the PTAM first travels up into the die stack, and the dies' DfT resources such as the DWR and 2D core-internal scan chains are accessed for test only on the PTAM's way down the stack to the external stack I/Os. *Test@First* and *Test@Last* dies can be freely combined in a single-tower die stack.

will tie these output-enable signals to logic '1' and allows the two corresponding configuration bits to be removed from the Configuration Register (Lines 006 and 007).

Following the flow in Figure 2, we have built a demonstrator software tool that parses FPP specifications in `.fpp` format, performs various semantic checks (as outlined in Section 5), and subsequently generates a corresponding Verilog gate-level netlist [30] for the specified FPP. We have used this demonstrator tool on the `taf.fpp` specification of Figure 5. The schematic view of the generated netlist in Cadence' Genus [31] is shown in Figure 6.

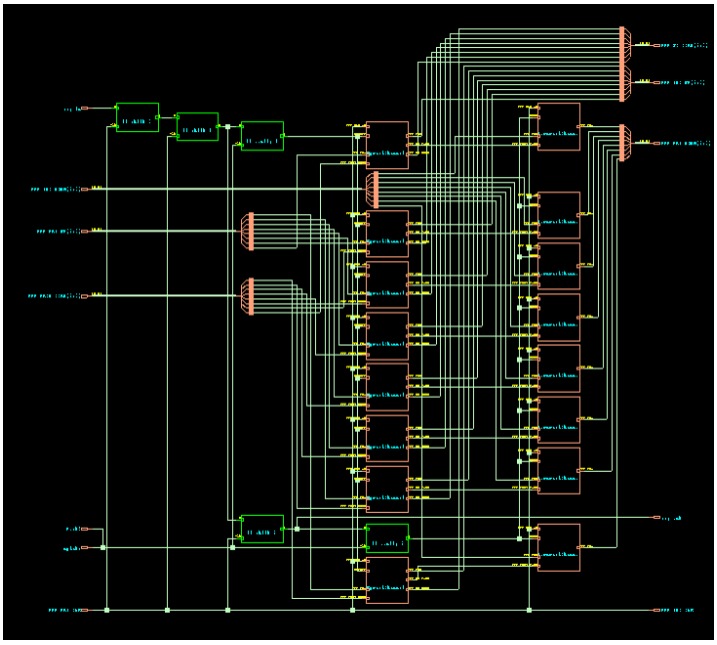

Figure 6: Schematic view of the generated Verilog netlist of the FPP example.

7 Conclusion

This paper described the FPP and its role in standard-under-development IEEE Std P1838. The paper proposed an FPP specification language (`.fpp`) using Google's Protocol Buffers. We provided the `fpp.proto` definition of the language, which could be released as part of the standard. With this definition, PB Compiler can automatically generate data access routines for multiple compute platforms and programming languages. These routines serve as building blocks in the construction of application software. As demonstrator, we made an FPP-to-Verilog generator tool. The paper closes with a realistic example FPP, for which we used the demonstrator software tool.

Acknowledgments

The authors thank the members of the IEEE Std P1838 Working Group for many fruitful discussions, in particular Tiger Teams 3 and 5 on resp. the FPP and Description Languages. We thank Patrick De Ryck of KU Leuven's Faculty of Engineering Technology for his supervision of the MSc graduation project of Yu Li and Ming Shao on the topic of this paper.

References

[1] Rick Merritt. Roadmap Says CMOS Ends ∼2024; IRDS points to chip stacks, new architectures. *EE Times*, March 23, 2017. `https://www.eetimes.com/document.asp?doc_id=1331517`.

[2] Eric Beyne. The 3-D Interconnect Technology Landscape. *IEEE Design & Test*, 33(3):8–20, June 2016. doi:10.1109/MDAT.2016.2544837.

[3] Shichun Qu and Yong Liu. *Wafer-Level Chip-Scale Packaging*. Springer-Verlag, New York, NY, USA, 2015. doi:10.1007/978-1-4939-1556-9.

[4] Kirk Saban. *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency*. Xilinx, October 2010. White Paper, see `http://www.xilinx.com/support/documentation/white_papers/wp380_Stacked_Silicon_Interconnect_Technology.pdf`.

[5] Denis Detoit et al. A 0.9 pJ/bit, 12.8 GByte/s WideIO Memory Interface in a 3D-IC NoC-Based MPSoC. In *Symposium on VLSI Technology*, June 2013. `http://ieeexplore.ieee.org/document/6578712/`.

[6] Michael Alfano et al. Unleashing Fury: A New Paradigm for 3-D Design and Test. *IEEE Design & Test*, 34(1):8–15, February 2017. doi:10.1109/MDAT.2016.2624284.

[7] Hao Chen, Hung-Chih Lin, and Min-Jer Wang. Fan-Out Wafer Level Chip Scale Package Testing. In *Proceedings IEEE International Test Conference Asia (ITC-Asia)*, September 2017.

[8] Erik Jan Marinissen. Challenges and Emerging Solutions in Testing TSV-Based 2.5D- and 3D-Stacked ICs. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 1277–1282, March 2012. doi:10.1109/DATE.2012.6176689.

[9] Mukesh Agrawal and Krishnendu Chakrabarty. Test-Cost Modeling and Optimal Test-Flow Selection of 3-D-Stacked ICs. *IEEE Transactions on Computer-Aided Design*, 34(9):1532–1536, September 2015. doi:10.1109/TCAD.2015.2419227.

[10] Mottaqiallah Taouil et al. Using 3D-COSTAR for 2.5D Test Cost Optimization. In *Proceedings IEEE International Conference on 3D System Integration (3DIC)*, pages 1–8, October 2013. doi:10.1109/3DIC.2013.6702351.

[11] Erik Jan Marinissen and Yervant Zorian. IEEE 1500 Enables Modular SOC Testing. *IEEE Design & Test*, 26(1):8–16, January/February 2009. doi:10.1109/MDT.2009.12.

[12] Erik Jan Marinissen et al. Test Access Architecture for TSV-Based 3D Stacked ICs, Priority date: September 20, 2010; Patent granted: January 19, 2016. US Patent 9,239,359 (B2), `https://patents.google.com/patent/US9239359`.

[13] Erik Jan Marinissen, Jouke Verbree, and Mario Konijnenburg. A Structured and Scalable Test Access Architecture for TSV-Based 3D Stacked ICs. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 269–274, April 2010. doi:10.1109/VTS.2010.5469556.

[14] Erik Jan Marinissen et al. 3D DfT Architecture for Pre-Bond and Post-Bond Testing. In *Proceedings IEEE International Conference on 3D System Integration (3DIC)*, November 2010. doi:10.1109/3DIC.2010.5751450.

[15] Erik Jan Marinissen et al. A DfT Architecture for 3D-SICs Based on a Standardizable Die Wrapper. *Journal of Electronic Testing: Theory and Applications*, 28(1):73–92, February 2012. doi:10.1007/s10836-011-5269-9.

[16] Sergej Deutsch et al. DfT Architecture and ATPG for Interconnect Tests of JEDEC Wide-I/O Memory-on-Logic Die Stacks. In *Proceedings IEEE International Test Conference (ITC)*, pages 1–10, November 2012. doi:10.1109/TEST.2012.6401569.

[17] Chen-An Chen et al. Cost-Effective TAP-Controlled Serialized Compressed Scan Architecture for 3D Stacked ICs. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 107–108, November 2013. doi:10.1109/ATS.2013.29.

[18] Yassine Fkih et al. 2D to 3D Test Pattern Retargeting Using IEEE P1687 Based 3D DFT Architectures. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 386–391, July 2014. doi:10.1109/ISVLSI.2014.83.

[19] Saman Adham and Erik Jan Marinissen. IEEE P1838 Web Site. `http://grouper.ieee.org/groups/3Dtest/`.

[20] Erik Jan Marinissen, Teresa McLaurin, and Hailong Jiao. IEEE Std P1838: DfT Standard-under-Development for 2.5D-, 3D-, and 5.5D-SICs. In *Proceedings IEEE European Test Symposium (ETS)*, May 2016. doi:10.1109/ETS.2016.7519330.

[21] IEEE Standards Association. *IEEE Std 1149.1™-2013, IEEE Standard for Test Access Port and Boundary-Scan Architecture*. IEEE, May 2013. doi:10.1109/IEEESTD.2013.6515989.

[22] IEEE Standards Association. *IEEE Std 1500™-2005, IEEE Standard Testability Method for Embedded Core-based Integrated Circuits*. IEEE, August 2005. doi:10.1109/IEEESTD.2005.96465.

[23] Chun-Chuan Chi et al. DfT Architecture for 3D-SICs with Multiple Towers. In *Proceedings IEEE European Test Symposium (ETS)*, pages 51–56, May 2011. doi:10.1109/ETS.2011.52.

[24] Christos Papameletis et al. A DfT Architecture and Tool Flow for 3D-SICs with Test Data Compression, Embedded Cores, and Multiple Towers. *IEEE Design & Test*, 32(4):40–48, July/August 2015. doi:10.1109/MDAT.2015.2424422.

[25] Ozgur Sinanoglu et al. Test Data Volume Comparison of Monolithic Testing vs. Modular SOC Testing. *IEEE Design & Test*, 26(3):25–37, May/June 2009. doi:10.1109/MDT.2009.65.

[26] Konstantin Shibin et al. At-Speed Testing of Inter-Die Connections of 3D-SICs in the Presence of Shore Logic. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 79–84, November 2015. doi:10.1109/ATS.2015.21.

[27] Erik Jan Marinissen et al. Vesuvius-3D: A 3D-DfT Demonstrator. In *Proceedings IEEE International Test Conference (ITC)*, October 2014. Paper 20.2.

[28] Google. Protocol Buffers, 1999. `https://developers.google.com/protocol-buffers/`.

[29] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. In *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky, MT, USA, October 2009. `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

[30] IEEE Standards Association. *IEEE Std 1364™-2005, IEEE Standard for Verilog Hardware Description*. IEEE, 2006. doi:10.1109/IEEESTD.2006.99495.

[31] Cadence Design Systems. *Genus GUI Guide*. 2017.