Terminology and Definitions for Multiported Functional Units.

David J. Greaves — Rough Draft November 2021/Sept 2023

***Abstract*—This note defines a generic framework and reference terms for formal specification of synchronous functional hardware units (FUs) to serve as a basis for compositional proof, automated system assembly (and partition over multiple FPGAs), incremental HLS and design reuse. It recommends a canonical description framework for net-level interfaces that is an extension of a TLM (transactional-level modelling) signature.**

This note defines formal terminology and notation for discussion of generic sub-components in digital logic. The following terms appear in bold font where they are defined: *FU, port, net-level, directorate, clock-enable, data net, handshake net, transaction, idle state, sequential consistency, exclusivity requirements, standard synchronous interface, fully-pipelined, re-initiation interval.*

We use the term **FU** (functional unit) for such a component and concentrate on synchronous (single clock) components only in this note. The aim is to describe a generic framework that encompasses the interfacing needs for the vast majority of FUs used in contemporary hardware designs.

I. FU (Functional Unit) and Ports

A functional unit **FU** is a component that could be as simple an AND gate but is more typically an ALU or RAM. Describing a full electronic data sheet for an FU is a basis for

- automatic wiring between them (either rule-directed or with glue logic synthesis[1]),
- incremental HLS where one compilation run deploys existing FUs.
- documentation,
- formal verification of conformance to protocols,
- design reuse (including verification infrastructure),
- adding-up non-functional metrics, such as power use and gate count.

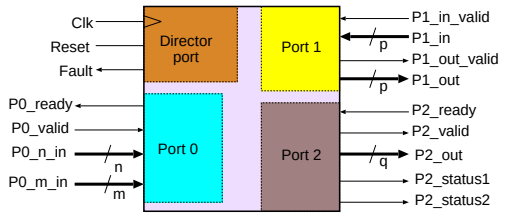

Fig. 1. A generic, synchronous FU, with three data ports and the obligatory director port.

Fig. 1 illustrates the general structure of a synchronous FU. A **synchronous FU** has only one clock input where all inputs are only looked at on the active edge of the that clock. The example is partition to show four sets of net-level connections. Each set corresponds to a **port**. We will later speak about the relationship between ports and **methods**. By **net-level** connection we mean a terminal that must be connected with a wire (called a net) to terminals on other components. Each terminal is either an input or an output terminal, and net-level interconnections obey a rule that exactly one output terminal is connected to each net.[1] This is a physical-layer design (consistency) rule.

One of the four ports is called the director port or the directorate. This connects nets that are part of the system management and which are intrinsically referenced by all the other ports. The **director** for a synchronous system contains the edge-triggered clock input (where the property that only its positive edge is significant is denoted with the triangle). Another net generally included in the directorate is a reset net (which can be synchronous or asynchronous and active high or active low). In this note, it is implied that the reset and clock are all fed from a common reset and clock generator unless specified otherwise (which we will not do in this note). A **CEN**, clock enable input may also be part of the directorate. When present, active clock edges where the CEN input is deasserted are ignored at all ports. Updates to internal state when CEN is deasserted do not typically occur, but some pipelined child FUs used inside the FU may not have CEN inputs and the FU must accommodate this by design (eg. a pipelined DSP multiplier in some families of FPGA). Other nets that make up the directorate in real systems include power supplies, fault indicator outputs and power sequencing controls for power saving — they are not discussed in this note.

The net-level connections within each port are normally grouped, as illustrated, into various control nets and data busses (although we shall not use these two terms henceforth.) In this document, apart from reverse-direction handshake nets, all nets in each such groups are all either inputs or outputs. In other words, our basic port is **simplex**.[2]. Where there are multiple logical busses (all travelling in the same direction) within a port, these are called, somewhat tatuologously, the **subfields** of the port.

In automata theory, the range of allowable values on a group defines a **symbol alphabet**. Its size is bounded by two-to-the number of nets in the group for binary logic. On the active edge of a clock, a group may or may not convey

[1] Tri-state nets are not considered in this note.

[2] Commonly, data needs to go in and out of an FU, so ports are typically grouped in either **put/get** or **put/aout** pairs, as explained later

a symbol according to the semantics of the port's protocol. A **protocol** specification defines exactly what sequence of symbols within a port conveys useful data over the port. Most protocols support **idle intervals** where nothing is being conveyed through the port.

Ports have a port type. Many components have multiple instances of a given port type. For instance, a multiplexor must have at least two 'input' ports and these may share a common type. To distinguish the ports, each has a **port instance name** (such as Port 1 and Port 2 in Fig. 1)

II. Standard Synchronous Interface (SSI)

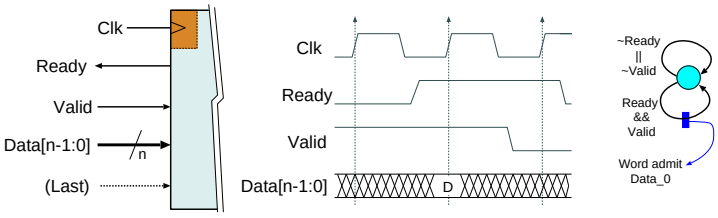

Fig. 2. Standard synchronous (simplex put) interface (SSI) port with one subfield: net-level view, timing diagram and interface automaton. Also shown is a minimal director port, consisting of just a clock input. (The last net indicates the last word of a multi-word transaction.)

The standard synchronous interface (SSI) is a very popular handshake protocol in everyday SoC design. However it cannot be trivally re-pipelined and hence **credit-based flow control** (CBFC) is also widely used. We shall augment this proposal to better cover CBFC in the future.

The SSI is generally simplex (moves data in one direction only), but we will also consider duplex variants. An input port is illustrated in Fig. 2. A handshake net runs in each direction. The timing diagram, in the centre, shows words are transferred over the data subfield on the positive edge of any clock cycle where both handshake nets hold. For an output port, the directions of all nets in the port (ie. not the clock, which remains an input) are reversed. This is a common design paradigm in contemporary hardware: five instances of this basic interface are used in the AXI protocol, one for each of its five 'channels'. (The interface automaton can be used for synthesis of glue logic in companion work.)

III. Transactional Port

We are primarily interested in transactional ports. The SSI is one such. A transactional port begins life (after reset) in an idle interval. For all practical ports, idle intervals can be detected by a predicate in the form of a temporal regular expression (eg. a SERES in PSL). For the standard synchronous interfaces (SSIs) described later, there is an idle interval on any clock edge where either `valid` or `ready` is deasserted. For the the rest of the port's life, the port conveys a sequence of transactions, separated by actual or nominal idle intervals. The idle interval can be nominal for two reasons. The first is back-to-back transactions, where one one transaction follows immediately after the one before. The second arises when there is a gap, but it is not simultaneous over all groups in the port since the protocol spans in terms of clock cycles is greater than or equal to the initiation interval (II) defined later.

A **transactional port** conveys a logically grouped number of values as one unit of operation. For instance, for a RAM, the values that need to be conveyed as part of a write transaction are the target address, the data to be stored and perhaps some byte lane qualifiers. To read a RAM, either a single or two transactions are required. Using separate transactions, one to convey the address and receive the result, is often appropriate, especially where the RAM is accessed over a network-on-chip. (This is known, in the Bluespec community, as the **put-get paradigm**; BlueParrot uses **put/aout** instead.)

Transactional ports can be readily simulated at a level of abstraction above the net level using transactional-level modelling (TLM). In TLM modelling, the values conveyed over a port are modelled with a method call on an OO instance that represents the FU. The nets that carry data are modelled in the arguments and return value to the method. The nets that provide handshaking in the hardware implementation do not appear at the TLM level, being replaced with the mechanics of subroutine call.

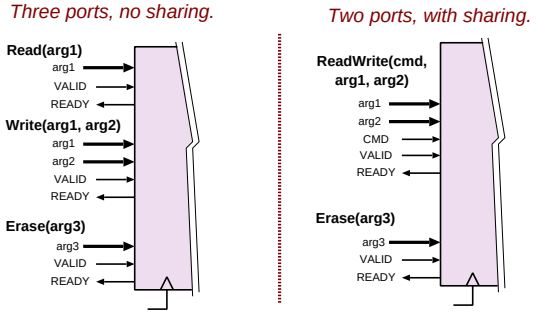

Fig. 3. Port sharing: two variations of the same FU port. The method name to `CMD` subfieldencoding is descriped in the accompanying IP-XACT.

In our formalism, several methods may be invokable on one port (or port pair) using **port sharing**. The subfields of the port then become the union of the subfields of the sharing methods. A further command subfield is added to the put side of the port which carries a method enumeration type whose binary coding is defined in the accompanying IP-XACT. A constraint arises which is that all of the arguments to the various methods sharing a port, where they have formal parameters with the same name, such as `address_bus`, need that formal to have the same type (eg. bit width).

Certain bus protocols support 'multiple outstanding transactions', especially those that connect a DRAM subsystem to a processor. For instance, a CPU may have multiple load-store stations that each issue a tagged memory request and remain blocked until a result with that tag is returned. The results can be out-of-order. But that meaning of 'trans-

action' is above the meaning in this document, with each individual request and response that is conveyed over the port and interconnect being considered a transaction.

Hardware ports are considered independent: the operations on one port of an FU can, in principle, be performed without reference to the state of other ports. **Sequential consistency** concepts will typically still apply at a higher level, but are beyond the scope of the definitions here. For instance, a multi-ported register file will maintain all standard RaW-like dependencies between its read and write ports, but precisely when written data becomes visible at a write port is outside the low-level specification of the ports themselves.

Where some combinations of simultaneous transactions at different ports cannot be supported by an FU, the approach here is to insist the exclusivity matrix has a clique structure and use one physical port per clique. This removes physical-layer **exclusivity requirements** but requires a trivial shim in the TLM to physical-layer mapping, where two or more methods are combined into transactions on a physical port by adding an extra `cmd` symbol, which enumerates which operation is invoked. Given that only one transaction is current (see out-of-order note above) at any one port at any one time, this guarantees exclusivity.[3]

IV. Put/Get, Put/Aout and Deadlock

Simplex ports may be grouped into pairs or larger groups to define a complete 'bus'. For instance, AXI has 5 simplex ports (write data, write address, write response, read address and read response).

Bluespec commonly uses the Put/Get paradigm (which is a variant of the mailbox paradigm). There are two simplex ports. The put side sends commands or requests into an instantiated server. The get side actively collects the results. Both ports use the standard synchronous interface (SSI) with the valid signal (called `EN`) as an input signal to the child/server component on both ports. Therefore the flows in the reverse direction to the valid signal on a get-style port.

BlueParrot uses the Put/Aout paradigm (which is a variant of a FIFO or BUFFER paradigm). The Put side is the same as Bluespec, with the data subfields being input to the child/server component, but the Aout side has the `valid` signal coming out of the child/server component and so remains consistently flowing in the same direction as the data subfields it is qualifying.

You might argue that, for the standard synchronous interface, since only the conjunction of the `ready` and `valid` signals is significant, and since all these simplex variants have one signal in each direction, the difference between Put/Get and Put/Aout is simply a matter of handshake net naming. This is true to a certain extent, but for deadlock checkers to comprehend sequential dependencies between handshake nets, where renaming is needed to make a port correspond to our recommended taxonomy, the changes need also need to be reflected in the dependency matrices. Moreover, the default rule where no matrix is given is that `valid` (out) should not depend on `ready` (in).

NB: AXI is here classified as two independent port groups with form (put/put/get) and (put/get).

[3] A TLM model may need to ensure calls are not re-entrant using locks or correct for timing using a quantum keeper.

V. Pipelined, Tokenised and Credit-based Flow Control

Figure 4 presents the three main forms of handshaking used across the industry today. In the centre, the dual-simplex, tokenised FU is shown. This has a Put/Aout pair of standard synchronous interfaces. Both are simplex (subfield data only moving in one direction) with the data direction being the same as the `valid` signal.

The **generic duplex** version of one of these ports (not shown) would have data moving in both directions, but qualified under the same handshake.

Our formalism of the standard synchronous interface allows either or both of the handshake nets to be specified as missing, in which case it implicitly always holds.

Another aspect of the formalism is **subfield time offsets**. When missing for a subfield, the data is qualified by the handshake during the clock cycle they denote (ie. one where they both hold). Where present, it defines that the subfield data is valid that number of (clock-enabled) clock cycles before or after. Negative values denote valid before handshake cycle. This facility, together with the support for missing handshakes, enables us to fold pipelined FUs into our formalism, as explained in the next paragraph.

On the left, a fully-pipelined FU is shown. This has no handshake nets and accepts a new argument on every clock cycle. The result is delivered on its output port some fixed-number of clock cycles later, known as its latency, here illustrated as 5 cycles. Where the initiation interval is not unity, or for energy saving by skipping the processing of idle arguments, at least one handshake net needs to be added to this baseline component. Therefore, our recommended model for this component is a degenerate form of our generic duplex component. Any vestigial standard synchronous handshake nets (just `VALID` for energy saving is illustrated) provide a reference time frame. The illustrated `result` output is not considered part of a handshake-free second port to the FU. Instead, it is considered a subfield of the left-hand port 0, with backwards data direction and with subfield time offset of +5 cycles.

On the right of Figure 4, for completeness, we illustrate the **credit-based flow control** (CBFC) paradigm. This FU has both ports credit controlled, although a commonly encountered FU is the bridge from or to CBFC to SSI. Under CBFC, the `ready` reverse direction signal is replaced with a credit return net. This is not synchronised with the forward simplex data and its `valid` net. It operates independently. See [2].

VI. Conclusions: Updated August 2023

Electronic data sheets for FUs need to be augmented with a description of the handshake protocol semantics. As a

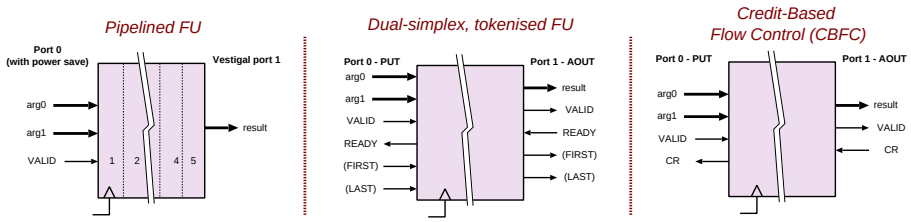

Fig. 4. Three general forms of handshake: pipelined (no handshake), standard-synchronous with put/aout paradigm and credit-based.

proof of concept, the authors have developed BlueParrot, a prototype HLS synthesiser that compiles a TLM-callable method body into gates, where the method body invokes TLM calls on pre-compiled FUs and leaf cells [citation to be provided].

Many other aspects of an FU are also documented, including whether it has side effects, whether it is freely replicated for load balancing, whether it needs access to main DRAM memory space and whether it has sequential dependencies between its handshake nets for deadlock avoidance.

The current implementation and definition of the IP-XACT extensions is being described on the following web page and in the src files for the HPR L/S logic synthesiser library.

`https://www.cl.cam.ac.uk/~djg11/cards.html`
`https://bitbucket.org/djg11/bitbucket-hprls2/src/master/`

We hope to see the community adopt a set of IP-XACT extensions for documenting these aspects.

REFERENCES

[1] M. N. David J Greaves, "Synthesis of glue logic, transactors, multiplexors and serialisors from protocol specifications," *IET Conference Proceedings*, pp. 171–177(6), January 2010. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/ic.2010.0148

[2] D. J. Greaves, *Modern SoC Design on Arm*. www.arm.com/resources/education: Arm Educational Media, 2021.