ESP Accelerator Specifications

System Level Design Group

July 12, 2021

Contents

1. Introduction

This document describes the signal-level protocol specification of an ESP accelerator. The guide is intended for RTL designers who wish to implement a native ESP accelerator using a hardware-description language, such as SystemVerilog, or VHDL. Any accelerator that complies with the protocol specification described in this guide can be integrated in ESP and leverage all platform services through the *ESP accelerator socket*.

This document does not describe the *ESP third-party accelerator flow*. The latter enables the seamless integration of an existing accelerator IP leveraging an ARM AMBA open standard interface.

1.1 Conventions

- **bitwidth:** number of bits. This is typically associated to a signal, or to a unit of data.
- **token:** the unit of input or output data transferred between the accelerator and the ESP socket. The bitwidth of a token depends on the particular accelerator and may vary across different transactions over a bus or data channel.
- **beat:** the unit of data transferred on a bus, or a data channel. The bitwidth of one beat depends on the particular implementation of the accelerator (e.g. *dma32* or *dma64*) and not on the data type of the input or output token in a transaction. Therefore, for any given implementation of an ESP accelerator, the bitwidth of a beat is constant.
- **flit:** the unit of data transferred over a network-on-chip (NoC). For ESP accelerators, the bitwidth of a flit is equal to the bitwidth of a beat plus two bits. These additional bits indicate if the flit is the head, part of the body, or the tail of a packet.
- **packet:** a set of flits transferred in an ordered sequence across the NoC. Packets must have one header flit, one tail flit and as many body flits as necessary. Single-flit packets have just one flit with both *head* and *tail* bits set. A packet that is granted a link of the NoC will traverse such link from head to tail not interleaved with another packet.
- **initiator** or **master:** a component that can initiate a transaction over a bus, or a NoC.
- **target** or **slave:** a component that servers a transaction initiated by a master.
- **latency-insensitive channel (LIC):** a bundle of data wires and two control wires named *ready* and *valid*. During read transactions, the master drives the *ready* control signal, while the slave drives the data and the paired *valid* control signal. Roles are inverted for write transactions. A beat is transferred over a LIC when both *ready* and *valid* are set. Both master and slave have the ability to delay the transfer of a beat for as many cycles as necessary. For more on latency-insensitive channels please refer to [Carloni, 2015].
- **CSR:** configuration and/or status register.
- **DMA:** the acronym for *direct-memory access*. When referring to an ESP accelerator, the term DMA refers to the mechanism used by the accelerator to access data in the system memory hierarchy. A DMA transaction initiated by an accelerator in ESP may be accessing external memory *directly* or by mediation of the ESP cache hierarchy. The selection is managed by software at run time and is transparent to the accelerator.
- **PLM:** the accelerator's private local memory, composed of a set of SRAM bank groups customized for the accelerator's datapath.

2. ESP Accelerator Specifications

2.1 Accelerator Model

The block diagram of Figure 2.1 illustrates the ESP accelerator socket and shows the three main set of signals at the interface of an ESP accelerator: read and write port for data transfers through DMA requests, configuration port and interrupt line.

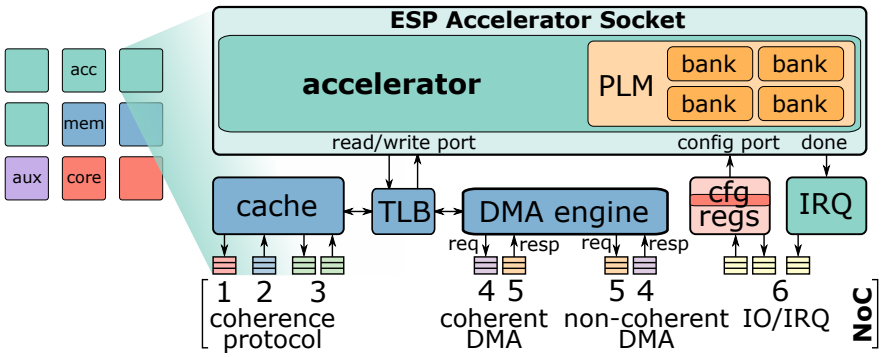

Figure 2.1: Block diagram of ESP accelerator tiles

A typical ESP accelerator is composed of three control blocks (*configuration*, *load*, *store*), one or more computation blocks and a customized private local memory (PLM).

Once configuration registers are valid, the configuration block activates the other components. The *load* module initiates the first DMA transaction to fetch the input data, or a portion of it, from the system memory hierarchy into the PLM. Next, the computation blocks process the available input and produce the corresponding output. Finally, the *store* block writes back the output to the system memory hierarchy with a DMA request. A single accelerator invocation from software typically results into multiple iterations of *load*, *compute* and *store* phases, therefore we recommend implementing a portion of the PLM as a set of ping-pong buffers to enable pipelining. Depending on the particular task and accelerator implementation, this strategy may significantly improve the overall accelerator throughput by masking most of the time for data transfers with the overlapping computation steps.

The above accelerator model corresponds to what ESP automation provides through any of the available high-level synthesis flows. RTL designers are not required to follow these directions, as long as they comply with the signal-level protocol at the interface with the ESP socket.

2.1.1 Accelerator Configuration

The configuration block regulates the accelerator execution and implements the interface with software by sampling the value of common and user-defined configuration registers located in the ESP socket.

Table 2.1: Description of the ESP accelerator configuration port.

Signal	Driver	Description
`clk`	socket	accelerator clock.
`rst`	socket	accelerator reset **active low**. The socket activates this reset signal when software clears the interrupt request to ensure that the accelerator is ready for a new invocation and internal state is clean. If the accelerator is expected to retain its state across different invocations, a user-defined configuration register can be used to implement a software-controlled reset signal.
`conf_done`	socket	Configuration registers are valid and computation can start. This signal is active high and asserted for one `clk` cycle to trigger the accelerator execution.
`conf_info_<register_name>`	socket	User-defined configuration input. The corresponding memory-mapped configuration registers are automatically generated in the ESP socket when creating the SoC instance. There can be up to 14 user-defined registers that must be listed in the accelerator definition XML file. For each register the accelerator must expose one `conf_info_` input. Bitwidth must be between 1 and 32 bits. These inputs should be considered valid when the `conf_done` input is active high.
`acc_done`	accelerator	Single-cycle pulse active high. This flag indicates that the accelerator has completed its task. The pulse should occur only after the last DMA write transaction has completed and all output data have been transferred from the PLM to the memory hierarchy. Asserting `acc_done` will trigger an interrupt request to the interrupt controller located in the ESP auxiliary tile. The software interrupt handler is responsible for clearing the interrupt, thus resetting the state of the socket and activating the `rst` input of the accelerator.
`debug`	accelerator	32-bit debug output. The accelerator designer can use this output to encode error codes. The state of this output can be accessed through the common memory-mapped registers present in the socket.

2.1.2 Private Local Memory

The PLM can be generated with the ESP `Memgen` utility, which combines SRAM primitives available as part of the target technology libraries. Alternatively, the accelerator designer can manually implement the PLM in RTL.

The PLM is not memory mapped, hence it is not exposed to software. Furthermore, the PLM is not part of the SoC cache hierarchy as it is solely intended as a customized working buffer for the accelerator data path. As a result, the PLM has no external interface exposed to the ESP accelerator socket and RTL designers are not required to comply with any hierarchy convention or signal-level protocol to implement the PLM, unless they use the ESP `Memgen` utility

(refer to the ESP Memory Generator documentation for further information). Accelerators that operate on small batches of data and don't have particular buffering requirements can also be implemented without a PLM.

2.2 DMA Transactions

The master of a direct-memory access (DMA) transaction is always an accelerator. The accelerator initiates a DMA read transfer through the dma_read_ctrl channel and a DMA write transfer through the dma_write_ctrl channel. Tables 2.2 and 2.3 describe the fields and the encoding of the two control channels.

DMA control channels are LIC that follow a simple protocol [Carloni, 2015]: when both *valid* and *ready* control signals are set, the value of the data bus is sampled by the slave. From the accelerator view point *valid* and *ready* are independent and there should be no combinational path between the two signals.

Table 2.2: Encoding of DMA size

Encoding	Name	Bitwidth
000	BYTE	8
001	HWORD	16
010	WORD	32
011	DWORD	64

An ESP accelerator does not issue requests using physical addresses. The field *index* of the control channels indicates an offset with respect to a virtual memory region reserved for the accelerator. The ESP device driver allocates this region in virtual pages and generates a corresponding page table. The ESP accelerator socket handles address translation, therefore the accelerator can operate as if the reserved area was contiguous.

Table 2.3: Description of the DMA control channels *dma_read_ctrl* and *dma_write_ctrl*.

Signal	Driver	Description
dma_[read|write]_ctrl_data_index	accelerator	Offset of a DMA read or write transaction expressed as number of beats. This offset is used to compute the starting address of the transaction.
dma_[read|write]_ctrl_data_length	accelerator	Length of a DMA read or write transaction expressed as number of beats.
dma_[read|write]_ctrl_data_size	accelerator	Bitwidth of the data token for the DMA transaction. This signal is used to correct the NoC flits when the processor architecture follows the *big endian* convention to store data in memory. This signal follows the encoding in Table 2.2.
dma_[read|write]_ctrl_valid	accelerator	Flag indicating a new DMA transaction request. When set, all data fields must be valid. This flag must not depend combinationally on the corresponding *ready* signal.
dma_[read|write]_ctrl_ready	socket	Flag indicating that the ESP socket is ready to accept a new DMA request. This flag must not depend combinationally on the corresponding *valid* signal.

For an accelerator with N-bits DMA interface (e.g. 64 bits), the physical address in bytes of a DMA transaction is computed by the ESP socket as follows:

$$addr = walk\_accelerator\_ptable(index * N/8) \tag{2.1}$$

The user-level driver is responsible to prepare data in memory using the same offsets used for DMA transfers by the accelerator. Offset calculation can be defined at design time by hard-coding the logic to compute offsets in the accelerator. Alternatively, offsets can be computed in software and configured at run time through user-defined control and status registers (CSRs).

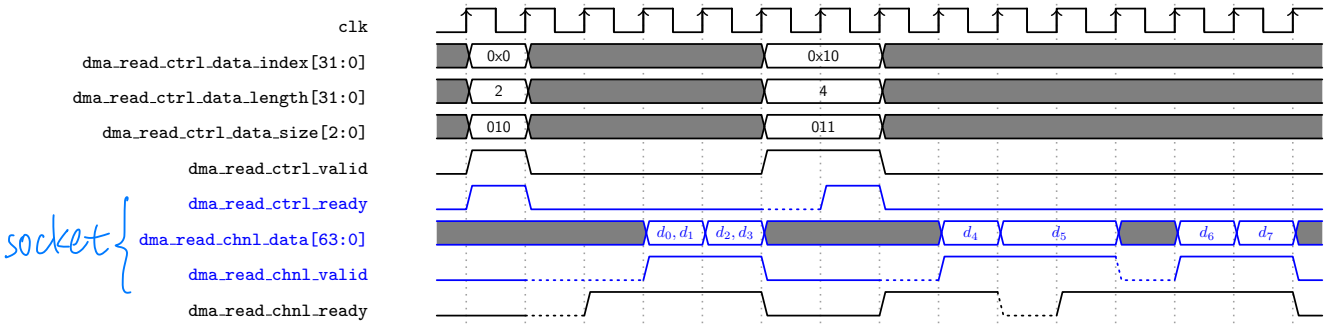

Figure 2.2: Example of a DMA read transaction. Signals driven by the ESP socket are marked in blue.

A DMA transaction is initiated with a single beat transfer on the DMA control channels. Once this transfer completes on the read control channel, the accelerator waits for the socket to fetch the requested data by setting *ready* high on the DMA read channel. A beat is successfully transferred any time *ready* and *valid* are set high during the same cycle. There is no restriction on the throughput of the transfer: the accelerator can apply backpressure by de-asserting the *ready* signal at any time. However, the accelerator must eventually complete the transaction by receiving exactly the number of beats requested. Early termination will cause a deadlock condition of the socket. Depending on the length of the transfer, deadlock can propagate to the NoC and even to a memory tile.

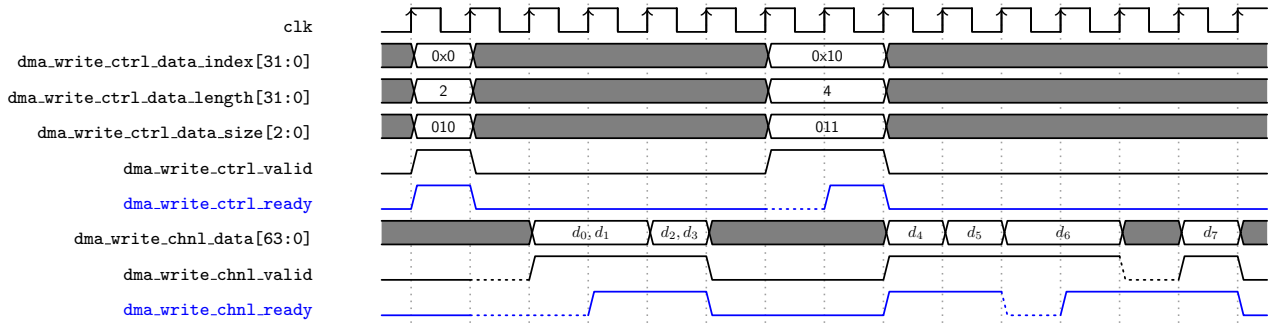

Figure 2.3: Example of a DMA write transaction. Signals driven by the ESP socket are marked in blue.

Symmetrically, when a DMA read transfer is configured, the accelerator must transfer the exact number of beats set with the *length* field. Data beats are transferred through the DMA write channel by setting the *valid* flag high when the corresponding data signal is valid. A beat is transferred when both *valid* and *ready* are set during the same cycle. No restriction is imposed on the throughput of the transfer. The accelerator must hold valid data on the DMA write channel when the socket is not ready to sample it. This condition may occur in case of contention for NoC links, or external memory channels.

Figure 2.2 and 2.2 show two examples of DMA read and DMA write transactions. Signals in blue are driven by the socket, while signals in black are driven by the accelerator. Dotted lines indicate back-pressure, which can be applied by either the accelerator or the socket.

Bibliography

[Carloni, 2015] Carloni, L. P. (2015). From latency-insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151.