NETCOPE P4 COMPILER

User's Manual

Netcope Technologies, a.s.

Contents

1 Overview

Netcope P4 Compiler is accessible via the Netcope P4 Cloud service at `np4.netcope.com`. This User's Manual describes how P4 programs should be written by the user, so that they are succesfully compiled by the Netcope P4 Compiler.

2 Supported Features

2.1 P4 Actions

The following action keywords are supported by the Netcope P4 compiler:

Action	Supported from ver.	Description
`add_header`	1.0	Adds the protocol header.
`copy_header`	1.0	Makes a header copy.
`modify_field`	1.0	Modification of the field value.
`add_to_field`	1.0	Adds a value to the to the field.
`subtract_from_field`	1.2	Subtracts a value from the field.
`modify_field_with_hash_based_offset`	1.1	Initiate the computation of a hash value from the header set. The result is then used for the offset computation.
`bit_and`	1.2	Logical AND operation of two values.
`bit_or`	1.2	Logical OR operation of two values.
`bit_xor`	1.2	Logical XOR operation of two values.
`add`	1.2	Sums two values and stores it into the field.
`subtract`	1.2	Subtracts two values and stores it into the field.
`shift_left`	1.2	Bitwise left shift.
`shift_right`	1.2	Bitwise right shift.
`drop`	1.0	Dropping of output frame.
`no_op`	1.0	No oregister_readperation.
`register_read`	2.0	Read from the register.
`register_write`	2.0	Write to the register.

Table 2.1: *Supported P4 actions*

The following actions are currently not supported by Netcope P4. Ask your Netcope sales representative for a roadmap for future Netcope P4 features.

Action	Description
modify_field_rng_uniform	Generates the random values with a uniform distribution.
truncate	Shortening of output frame.
push	Each protocol header instances is moved down and the top of the stack is filled with the new protocol header.
pop	The top level header is removed and and all following headers are moved one position up.
count	Counter update.
execute_meter	Meter update.
generate_digest	Creates data for the software running on CPU. Exported data is stored in the intrinsic metadata digest field.
resubmit	Resends the packet with metadata to the parser.
recirculate	Sends the deparsed packet back to the input.
clone_ingress_pkt_to_ingress	Sends a copy of original packet to the ingress Match+Action.
clone_egress_pkt_to_ingress	Sends a packet copy from the egress Match+Action to the parser.
clone_ingress_pkt_to_egress	Sends a copy of original packet from the ingress Match+Action to the Buffer Mechanism.
clone_egress_pkt_to_egress	Sends a packet copy from the egress Match+Action to the Buffer Mechanism.

Table 2.2: *Unsupported P4 actions*

2.2 Other features

The list of non-action features highlights some of the main features of the Netcope P4 Compiler.

2.3 Other limitations and tips

Some of the implemented features and supported actions have limitations. These limitations derive from the limitations of used architecture and FPGA. Some of the main limitations are listed in a table below.

> Compiler is able to detect possible problems with the P4 code. It is therefore strongly advised to check the log of P4 to VHDL conversion before continuing with the synthesis.

Feature	From ver.
$P4_{14}$	1.0
Match-Action tables implemented as on-chip TCAMs	1.0
Netcope P4 C-language API	1.0
Generating FPGA bitstream for the NFB-100G2Q target	1.0
Generating FPGA bitstream for the NFB-200G2QL and SILICOM-FB2CGG3 targets	1.2
Generating netlists for Xilinx device families	1.1
Netcope P4 core identification (see chapter 4)	1.1
Match-Action tables (with exact matching) implemented as on-chip Cuckoo hashing	2.0
Basic packet multiplication to different physical interfaces	2.0

Table 2.3: *Supported P4 features*

Limitation	Description
General	
identifier names	Because of limitation of VHDL syntax some identifiers from P4 program are renamed by the compiler. This mainly means removing underscores from identifiers. Specifically all underscores from the beginning and end of identifiers are removed. Additionally any sequence of 2 or more underscores is replaced by only single underscore. For example identifier *___i_love____P4_* will be renamed to *i_love_P4*. To prevent any possible complications with the compilation or following configuration try to avoid these identifiers.
integer literals/constants	Integer literals and constants that require more than 32b are currently not supported. Avoid using such constants/literals in P4 source program. Instead split them into multiple 32b literals/constants.
Header fields	
variable length	Variable length of header fields is currently not supported. While these fileds will be parsed properly they can't be used in other stages (match+action, deparser).
Parsing	
similar parse states	Due to a known bug parsing of the same protocol within two parse states that are both accessed through different cases of the same *select* statement leads to incorrect behaviour. Avoid using such constructions and instead describe the same behaviour using multiple parse states and metadata setting.
Match+Action Tables	
range match type	Tables with range match type are not supported.
Hashes and checksums	
`modify_field_with_hash_based_offset`	Parameter *size* should be set to a constant value that is preferably power of 2. Setting it to any other value or parameter from enclosing action leads to ineffective implementation and resulting design will most likely be unable to meet FPGA timing constraints.
field_list	Field lists does not support the *payload* keyword. Field list can therefore consist only of header fields.
CSUM16 and *XOR16*	Computations of *CSUM16* or *XOR16* are carried out in paralel. This means that the result of such computations should not be used later within the same enclosing action. For example there should never be `modify_field_with_hash_based_offset(X,base,csum16_calc,size)` preceeding `add_to_field(X,value)`. To properly do this, create two tables applied one after another where the first table computes the hash and the second one then works with the result.

Limitation	Description
calculated_field	Calculated fields are currently disabled. The equivalent functionality can be achieved using a table and `modify_field_with_hash_based_offset` action.
efficient tables	If possible put any actions computing long *CSUM16* or *XOR16* into a separate table. This can lead to generation of more efficient HW design.
Registers	
`register_read`, `register_write`	Using and accessing registers in the P4 code can currently decrease the throughput of the architecture.
global registers and atributes	Global registers and attributes of registers (signed, saturating) are currently not supported.
multiple registers per Match+Action Table	Multiple register arrays tied to the same table will have their width changed to the maximum of their widths.
index of accessed register as action parameter	If the index of the accessed register is tied to action parameter, the width of the parameter will be 32b (regardless of the actual number of registers).
Architecture	
queues and buffers	The architecture currently does not support any type of queuing and buffering algorithms. The architecture consists of parser, ingress match+action tables and deparser.

3 Intrinsic Metadata

To accomodate for the target hardware platform and to save FPGA resources, P4 language `standard_metadata` is not supported. The compiler instead provides built-in intrinsic metadata defined as follows:

```
header_type intrinsic_metadata_t {
    fields {
        ingress_timestamp       : 64;
        ingress_port            : 8;
        egress_port             : 8;
        duplicate_en_mask       : 8;
        duplicate_dma0          : 8;
        duplicate_dma1          : 8;
        duplicate_eth0          : 4;
        duplicate_eth1          : 4;
        duplicate_eth2          : 4;
        duplicate_eth3          : 4;
        packet_len              : 16;
        hash                    : 4;
        user16                  : 16;
        user4                   : 4;
    }
}

metadata intrinsic_metadata_t intrinsic_metadata;
```

The items carry following information:

- `ingress_timestamp` – Time of packet reception. Upper 32 bits carry number of seconds since 1970-01-01, lower 32 bits carry number of nanoseconds within that second.
- `ingress_port` – Interface from which the packet was received. 0-127 are physical Ethernet RX interfaces, while 128-255 are software TX queues.
- `egress_port` – Interface to which the packet is to be sent. 0-127 are physical Ethernet TX interfaces, while 128-255 are software RX queues. This value is ignored if duplication was enabled.
- `duplicate_en_mask` – Mask used to enable packet duplication. If this mask is non-zero the packet duplication is enabled and *duplicate_** fields are used for forwarding instead of *egress_port*. Each bit of this field serves as a flag signalizing if the packet should be duplicated to the corresponding physical interface. Physical interfaces and their corresponding flags beginning from the LSB of this value are: PCI endpoint 0, PCI endpoint 1, physical network interface 0, physical network interface 1, physical network interface 2, physical network interface 3.
- `duplicate_dma0` – RX queue withing PCI endpoint 0 that the packet should be forwarded to. The value representing the queue is not global as in the case of *egress_port* but rather it is relative to this PCI endpoint.

- `duplicate_dma1` – RX queue withing PCI endpoint 1 that the packet should be forwarded to. The value representing the queue is not global as in the case of *egress_port* but rather it is relative to this PCI endpoint. For example when there are 2 PCI endpoints with 16 RX queues each then sending the packet through the RX queue 17 can be achieved by setting the second bit in *duplicate_en_mask* and setting the *duplicate_dma1* to 1.
- `duplicate_eth0` – TX interface withing physical network interface 0 that the packet should be forwarded to. The value representing the TX inteface is relative to the physical network port.
- `duplicate_eth1` – TX interface withing physical network interface 1 that the packet should be forwarded to. The value representing the TX inteface is relative to the physical network port. For example when there are 2 network ports with 4 TX interfaces each (this is the case for *_10G8 designs) then sending the packet through the TX interface 5 can be achieved by setting the fourth bit in *duplicate_en_mask* and setting the *duplicate_eth1* to 1.
- `duplicate_eth2` – TX interface withing physical network interface 2 that the packet should be forwarded to. There are currently no supported platforms with more than 2 network ports, this value can therefore be ignored.
- `duplicate_eth3` – TX interface withing physical network interface 3 that the packet should be forwarded to. There are currently no supported platforms with more than 2 network ports, this value can therefore be ignored.
- `packet_len` – Total packet length. In case of Ethernet frames, this number excludes Frame CheckSum, Preamble and InterFrame Gap.
- `hash` – Currently has no particular meaning and can be used for sending user data into SW (in the future it might be used for load balancing purposes).
- `user16` – 16 bits of user data. These data are sent to SW as a part of Netcope P4 header.
- `user4` – 4 bits of user data. These data are sent to SW as a part of Netcope P4 header.

This metadata provided by the compiler does not preclude from other metadata to be defined by the user, as long as there is no conflict in naming.

4 Netcope P4 Identification

The generated design has within itself a string that can identify what P4 source code was used to generate the Netcope P4 core. This string can be set to a specific value via P4 pragma statement. Pragma name to use is *core_identification*. Pragma statements in P4 are always tied to some object (entity) in the P4 description. However, this particular pragma can be tied to any of the used P4 objects and it will take effect. The identification can then be read out from the resulting design via Netcope P4 tool or Netcope P4 API. A simple example how to write such a pragma follows:

```
@pragma core_identification my_core_version_1
control ingress {
    ...
}
```

It is recommended to define this pragma and set the string to something unique for each version of a P4 program. This way it is possible to check if the correct version of Netcope P4 core is used within the design. A good example of such a string would be the project name followed by git revision number or a unique hash of the file.

5 FPGA resource usage

The Netcope P4 compiler generates firmware modules, which are then integrated with static modules to create a complete FPGA firmware. Depending on the target platform, some percentage of the FPGA logic resources are pre-occupied by these static firmware modules. Table 5.1 summarizes FPGA resource utilization for the supported platforms:

Target	Slice LUTs	Slice registers	BRAMs
NFB-100G2Q-100G1 (Xilinx Virtex-7 H580T)	126 562 (35%)	111 259 (15%)	257 (27%)
NFB-100G2Q-10G8 (Xilinx Virtex-7 H580T)	176 453 (49%)	198 183 (27%)	365.5 (39%)
NFB-200G2QL-100G2 (Xilinx Virtex UltraScale+ XCVU7P)	208 987 (27%)	210 533 (14%)	746 (52%)
NFB-200G2QL-10G8 (Xilinx Virtex UltraScale+ XCVU7P)	285 386 (36%)	309 691 (20%)	837.5 (58%)
FB2CGG3-100G2 (Xilinx UltraScale+ XCVU9P)	120 966 (10%)	134 478 (6%)	402 (19%)

Table 5.1: *Platform-specific resource utilization*

The FPGA resources consumed by the Netcope P4-generated module fully depends on the input P4 code. Table 5.2 provides some estimates for P4 language constructs that are the major contributions to FPGA chip resource consumption. Please keep in mind that even if the resource consumption of some of the constructs is pretty low, it might still be gated by the timming requirements.

P4 construct	Slice LUTs	Slice registers	BRAMs	Note
(Static parts of P4 module)	6962	9705	10	
IPv4 protocol parsing	1560	1738	0	
IPv4 protocol parsing x8	18587	10658	0	
IPv4 protocol parsing x16	53315	33351	0	
IPv4 protocol parsing x32	126804	131842	0	
8 Match-Action tables with single entry and 2 primitive actions	2336	1142	0	
16 Match-Action tables with single entry and 2 primitive actions	6024	2464	0	
32 Match-Action tables with single entry and 2 primitive actions	14356	5078	0	
64 Match-Action tables with single entry and 2 primitive actions	31064	12344	0	

P4 construct	Slice LUTs	Slice registers	BRAMs	Note
On-chip TCAM Match-Action table 16 items, 16b wide	120	125	0	
On-chip TCAM Match-Action table 16 items, 128b wide	372	770	0	
On-chip TCAM Match-Action table 128 items, 16b wide	512	577	0	
On-chip TCAM Match-Action table 128 items, 128b wide	1203	3326	0	
On-chip TCAM Match-Action table 1024 items, 32b wide	11290	8375	0	
On-chip TCAM Match-Action table 2048 items, 32b wide	21757	15707	0	Timming violation on NFB-100G2Q
On-chip Cuckoo Hashing Match-Action table 2048 items, 32b wide	977	603	6	
On-chip Cuckoo Hashing Match-Action table 2048 items, 64b wide	1100	661	12	
On-chip Cuckoo Hashing Match-Action table 2048 items, 128b wide	1776	1564	22	
On-chip Cuckoo Hashing Match-Action table 16384 items, 64b wide	1324	675	91	
On-chip Cuckoo Hashing Match-Action table 32768 items, 64b wide	2040	1007	183	
On-chip Cuckoo Hashing Match-Action table 65536 items, 32b wide	2203	1352	216	
On-chip LPM Match-Action table 1024 items, 32b wide	1720	2125	3	
On-chip LPM Match-Action table 1024 items, 128b wide	4612	5445	12	
On-chip LPM Match-Action table 2048 items, 32b wide	1779	2179	5.5	
On-chip LPM Match-Action table 2048 items, 128b wide	4814	5609	20	
On-chip LPM Match-Action table 8192 items, 32b wide	1885	2292	19	
On-chip LPM Match-Action table 16384 items, 32b wide	1972	2361	38	
On-chip LPM MAtch-Action table 32768 items, 32b-wide	2407	3006	75	

P4 construct	Slice LUTs	Slice registers	BRAMs	Note
On-chip LPM MAtch-Action table 65536 items, 32b-wide	2728	3605	149	

Table 5.2: *Resource utilization estimation*

Please note that these number are rough estimates, since the Netcope P4 compiler, as well as the subsequent FPGA synthesis tools, employ multiple optimization techniques to reduce the resource consumption. Also note that due to timing, routing and placement constraints, it is practically impossible to fully utilize all of the FPGA resources.

6 Netlist guide

6.1 Netlist Interface

The top level interface of the module (*NP4_ATOM*) has 5 main parts - input FLU interface (*RX_**), output FLU interface (*TX_**), input metadata interface (*IN_INTRINSIC_METADATA_**), output metadata interface (*OUT_INTRINSIC_METADATA_**) and MI32 configuration interface. Input FLU interface is used for sending in the data of packets for Netcope P4 processing. Input metadata should be valid along with the first part of each packet (during the clock cycle when *RX_SOP* is active). Metadata carrying the packet length (*IN_INTRINSIC_METADATA_PACKET_LEN_0*) has to be set properly for each packet and is updated within the Netcope P4 core automatically (for example when adding headers). All of the other metadata are transparent for the Netcope P4 core and does not need to be used (unless they were used in the P4 program that generated the Netcope P4 core). Similarly the output metadata are valid when *TX_SOP* is active. MI32 interface is used for configuring the M+A Tables of the Netcope P4 design (via Netcope P4 Atom tool and/or API). Definition of all of the signals from the interface is as follows:

```
-- Common interface
CLK             : in std_logic;    -- Clock
RESET           : in std_logic;    -- Synchronous reset

-- Input metadata (synchronized with first word of the packet - RX_SOP='1' and
-- RX_SRC_RDY='1' and RX_DST_RDY='1')
IN_INTRINSIC_METADATA_INGRESS_TIMESTAMP_0   : in std_logic_vector(63 downto 0);
IN_INTRINSIC_METADATA_INGRESS_PORT_0        : in std_logic_vector(7 downto 0);
IN_INTRINSIC_METADATA_EGRESS_PORT_0         : in std_logic_vector(7 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_EN_MASK_0   : in std_logic_vector(7 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_DMA0_0      : in std_logic_vector(7 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_DMA1_0      : in std_logic_vector(7 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_ETH0_0      : in std_logic_vector(3 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_ETH1_0      : in std_logic_vector(3 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_ETH2_0      : in std_logic_vector(3 downto 0);
IN_INTRINSIC_METADATA_DUPLICATE_ETH3_0      : in std_logic_vector(3 downto 0);
IN_INTRINSIC_METADATA_PACKET_LEN_0          : in std_logic_vector(15 downto 0);
IN_INTRINSIC_METADATA_HASH_0                : in std_logic_vector(3 downto 0);
IN_INTRINSIC_METADATA_USER16_0              : in std_logic_vector(15 downto 0);
IN_INTRINSIC_METADATA_USER4_0               : in std_logic_vector(3 downto 0);

-- Input interface (FLU)
RX_DATA         : in std_logic_vector(511 downto 0);
RX_SOP_POS      : in std_logic_vector(2 downto 0);
RX_SOP          : in std_logic;
RX_EOP_POS      : in std_logic_vector(5 downto 0);
RX_EOP          : in std_logic;
RX_SRC_RDY      : in std_logic;
RX_DST_RDY      : out std_logic;


-- Output interface (FLU)
TX_DATA         : out std_logic_vector(511 downto 0);
TX_SOP_POS      : out std_logic_vector(2 downto 0);
TX_SOP          : out std_logic;
TX_EOP_POS      : out std_logic_vector(5 downto 0);
TX_EOP          : out std_logic;
TX_SRC_RDY      : out std_logic;
TX_DST_RDY      : in std_logic;

-- Output metadata (synchronized with first word of the packet - TX_SOP='1' and
-- TX_SRC_RDY='1' and TX_DST_RDY='1')
OUT_INTRINSIC_METADATA_INGRESS_TIMESTAMP_0  : out std_logic_vector(63 downto 0);
OUT_INTRINSIC_METADATA_INGRESS_PORT_0       : out std_logic_vector(7 downto 0);
OUT_INTRINSIC_METADATA_EGRESS_PORT_0        : out std_logic_vector(7 downto 0);
```

```
OUT_INTRINSIC_METADATA_DUPLICATE_EN_MASK_0  : in std_logic_vector(7 downto 0);
OUT_INTRINSIC_METADATA_DUPLICATE_DMA0_0     : in std_logic_vector(7 downto 0);
OUT_INTRINSIC_METADATA_DUPLICATE_DMA1_0     : in std_logic_vector(7 downto 0);
OUT_INTRINSIC_METADATA_DUPLICATE_ETH0_0     : in std_logic_vector(3 downto 0);
OUT_INTRINSIC_METADATA_DUPLICATE_ETH1_0     : in std_logic_vector(3 downto 0);
OUT_INTRINSIC_METADATA_DUPLICATE_ETH2_0     : in std_logic_vector(3 downto 0);
OUT_INTRINSIC_METADATA_DUPLICATE_ETH3_0     : in std_logic_vector(3 downto 0);
OUT_INTRINSIC_METADATA_PACKET_LEN_0         : out std_logic_vector(15 downto 0);
OUT_INTRINSIC_METADATA_HASH_0               : out std_logic_vector(3 downto 0);
OUT_INTRINSIC_METADATA_USER16_0             : out std_logic_vector(15 downto 0);
OUT_INTRINSIC_METADATA_USER4_0              : out std_logic_vector(3 downto 0);

-- MI32 interface
DWR             : in std_logic_vector(31 downto 0);
ADDR            : in std_logic_vector(31 downto 0);
RD              : in std_logic;
WR              : in std_logic;
BE              : in std_logic_vector(3 downto 0);
DRD             : out std_logic_vector(31 downto 0);
ARDY            : out std_logic;
DRDY            : out std_logic
```

6.2 FrameLinkUnaligned

FrameLinkUnaligned (or FLU) is a high-performance synchronous point-to-point interface protocol. It is designed to be used with wide data buses (256 bits and above) without wasting as much bandwidth as the original FrameLink protocol.

6.2.1 Signals

FrameLinkUnaligned is composed of signals shown in the following table. All signals are synchronous to the CLK signal.

Signal name	Controlled b	Description
SOP	Sender	**Start of packet** indicates the clock cycle with the first data of a packet.
SOP_POS	Sender	**Packet start position** indicates the position of the first packet byte in the data word when SOP is active. Usually, DATA is 512 bits wide and the SOP_POS signal is 3 bits wide, then the value represents a multiple of 8 bytes. For example, the value "011" means that the first byte of the packet is at DATA(255 downto 192). A data word may be shared by two packets (indicated by the EOP and EOP_POS signals), in that case DATA(191 downto 0) may contain the end of a previous packet.
EOP	Sender	**End of packet** indicates the clock cycle with the last data of a packet.
EOP_POS	Sender	**Packet end position** indicates the position of the last packet byte in the data word when EOP is active. Usually, DATA is 512 bits wide and the EOP_POS signal is 6 bits wide. For example, the value "111110" means that the last byte of the packet is at DATA(502 downto 494). A data word may be shared by two packets (indicated by the SOP and SOP_POS signals), in that case DATA(511 downto 503) may contain the beginning of a next packet.
SRC_RDY	Sender	**Source ready** indicates that the Sender is ready to transmit data.
DST_RDY	Receiver	**Destination ready** indicates that the Receiver is ready to receive data.
DATA	Sender	Transferred data word of arbitrary width. In the NDK platform, only vectors whose width is power of 2 are used. Most common are 256 or 512 bit wide buses.

6.2.2 Data structure

FrameLinkUnaligned supports descending bit ordering. If n is the number of bits in the vector, then the bits are ordered in (n-1 downto 0) manner. The bit 0 is the most-significant bit and the bit n-1 is the least significant. Bytes are ordered in the same manner – the rightmost byte (bits 7 downto 0) is the least significant byte (LSB) and the leftmost byte is the most-significant byte (MSB).

6.2.3 Data transfer

The data transfer is controlled by the **Source ready** and **Destination ready** signals. When both the SRC_RDY and DST_RDY signals are active (equal to logic '1') a data word is transferred. The data transfer can be suspended for an arbitrary number of clock cycles by both the Sender or Receiver by disabling SRC_RDY or DST_RDY respectively (set to logic '0'). The SRC_RDY and DST_RDY signals have to be independent, i.e. they cannot be derived one from the other. The Sender must always assert SRC_RDY first and then wait for DST_RDY active. The Receiver must always assert DST_RDY first and

then wait for SRC_RDY active. The data bits as well as the frame control signals (SOP, SOP_POS, EOP and EOP_POS) are valid only if SRC_RDY is active.

6.2.4 Transfer examples

Transfer example 1

The time diagram in the Section 6.2.1 shows an example of transfer of a single packet over 512 bits-wide FrameLinkUnaligned bus. In the 2nd clock cycle, the Sender has prepared the first part of the packet for transmission. This is indicated by the assertion of SRC_RDY and SOP signals. The SOP_POS signal indicates the first byte of the packet. The Sender cannot send the data, because the Receiver is not ready (the DST_RDY signal is not active). The Receiver is ready in the 3rd clock cycle, so the first FLU word is transferred.

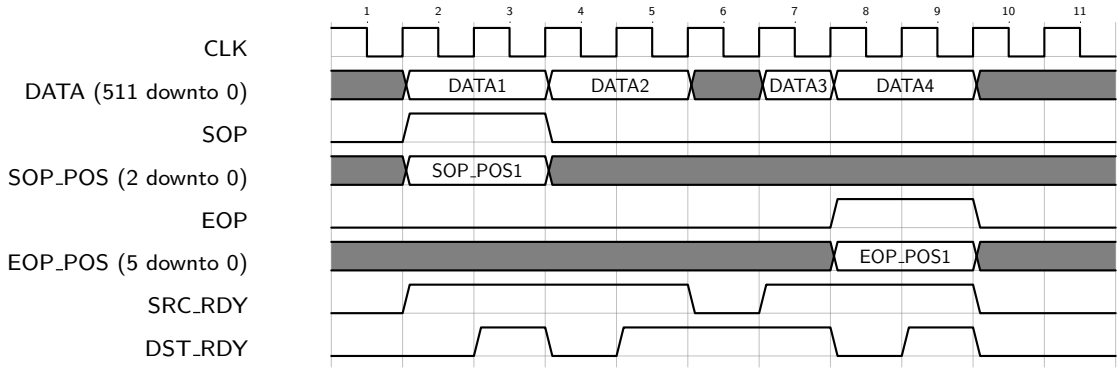

Figure 6.1: *FrameLinkUnaligned example communication 1*

The communication is suspended in the 4th clock cycle again, because the Receiver is not ready. The second data word is transferred in the 5th clock cycle. The communication waits in the 6th clock cycle, because the Sender has not prepared the data for transmission. The third data word is transferred in the 7th clock cycle, when the Sender becomes ready. The Sender has prepared the last data word in the 8th clock cycle. This is indicated by asserting the EOP signal. The EOP_POS signal indicates the last byte of packet. The last data word is transferred in the 9th cycle, when both the Sender and Receiver are ready.

Transfer example 2

The timing diagram in the Figure 6.2 shows an example of transfer of a few packets over 512 bits-wide FrameLinkUnaligned bus. In this example, the destination is always ready (the DST_RDY signal is active) so the Sender can transmit data all the time.

The communication starts in the 2nd clock cycle when the packet 1 with 64 bytes of length is transferred at once. The first byte of the packet is DATA(7 downto 0), therefore the SOP signal is asserted and the value of the SOP_POS signal is "000". The last byte of the packet is DATA(511 downto 504), therefore the EOP signal is asserted and the value of the EOP_POS signal is "111111".

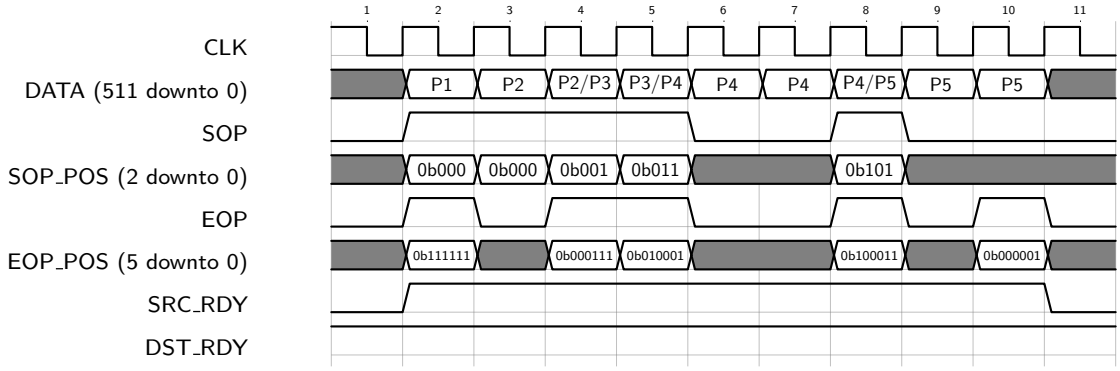

Figure 6.2: *FrameLinkUnaligned example communication 2*

The communication continues in the 3rd clock cycle when the first 64 bytes of the packet 2 with 72 bytes of total length are transferred. The first byte of the packet `DATA(7 downto 0)`, therefore the `SOP` signal is asserted and the value of the `SOP_POS` signal is `"000"`. The packet transfer is not finished in 3rd clock cycle, therefore the `EOP` signal is not asserted.

The remaining 8 bytes of the packet 2 are transferred in the 4th clock cycle. The last byte of the packet is `DATA(63 downto 56)`, therefore the `EOP` signal is asserted and the value of the `EOP_POS` signal is `"000111"`. The transfer of the packet 3 with 74 bytes of length starts in this clock cycle as well. The first byte of the packet is `DATA(71 downto 64)`, therefore the `SOP` signal is asserted and the value of the `SOP_POS` signal is `"001"`. 56 bytes of the packet 3 are transferred in the 4th clock cycle.

Clock cycle	Description
1	The sender is not ready
2	Transfer of packet 1 (bytes 0-63)
3	Transfer of packet 2 (bytes 0-63)
4	Transfer of packet 2 (bytes 64-71), transfer of packet 3 (bytes 0-55)
5	Transfer of packet 3 (bytes 56-73), transfer of packet 4 (bytes 0-39)
6	Transfer of packet 4 (bytes 40-103)
7	Transfer of packet 4 (bytes 104-167)
8	Transfer of packet 4 (bytes 168-204), transfer of packet 5 (bytes 0-23)
9	Transfer of packet 5 (bytes 24-87)
10	Transfer of packet 5 (bytes 88-89)
11	Transfer completed

The remaining 18 bytes of the packet 3 are transferred in the 5th clock cycle. The last byte of the packet 3 is `DATA(143 downto 136)`, therefore the `EOP` signal is asserted and the value of the `EOP_POS` signal is `"010001"`. The transfer of the packet 4 with 204 bytes of length starts in this clock cycle as well. The first byte of the packet 4 is `DATA(263 downto 256)`, therefore the `SOP` signal is asserted and the value of the `SOP_POS` signal is `"011"`. 40 bytes of the packet 4 are transferred in the 5th clock cycle.

64 bytes of the packet 4 are transferred in both the 6th and 7th clock cycle. In either of these clock cycles, the transfer is not started nor finished, therefore both `SOP_POS` and `EOP_POS` signals are

not active.

The remaining 36 bytes of the packet 4 are transferred in the 8th clock cycle. The last byte of the packet 4 is `DATA(287 downto 280)`, therefore the `EOP` signal is asserted and the value of the `EOP_POS` signal is `"100011"`. The transfer of the packet 5 with 90 bytes of length starts in this clock cycle as well. The first byte of the packet 5 is `DATA(327 downto 320)`, therefore the `SOP` signal is asserted and the value of the `SOP_POS` signal is `"101"`. 24 bytes of the packet 5 are transferred in the 8th clock cycle.

64 bytes of the packet 5 are transferred in the 9th clock cycle. In this clock cycle, the transfer is not started nor finished, therefore both `SOP_POS` and `EOP_POS` signals are not active.

Transfer of packet 5 is finished in the 10th clock cycle when remaining 2 bytes are transferred. The last byte of the packet 5 is `DATA(15 downto 8)`, therefore the `EOP` signal is asserted and the value of the `EOP_POS` signal is `"000001"`.

All packets have been transferred, therefore the `SOP` signal stays inactive.

6.3 MI32 Bus

The MI32 Bus (Memory Interface 32-bits) is a low speed communication bus intended to provide an access to firmware internal components from software user space (or in this case p4top component from other parts of firmware). It supports both read and write transactions. It is based on simple memory-like interface, which allows to connect user registers and memory blocks easily. Connected registers and memories are mapped into MI32 address space.

6.3.1 Signals

The MI32 bus is composed of signals shown in the following table.

Signal	Bit width	Owner	Operation	Description
`MI32_DWR`	31:0	Software	Write	Data to be written
`MI32_DRD`	31:0	Firmware	Read	Data to be read
`MI32_ADDR`	31:0	Software	Read and write	Address
`MI32_RD`		Software	Read	Read request
`MI32_WR`		Software	Write	Write request
`MI32_BE`	3:0	Software	Read and write	Byte Enable
`MI32_ARDY`		Firmware	Read and write	Address ready
`MI32_DRDY`		Firmware	Read	Data ready

All signals are synchronous to clock signal.

6.3.2 Write transaction

A write transaction is initiated by assertion of the `MI32_WR` signal representing a Write request. During the write transaction, the `MI32_DWR` data are written to the address according to `MI32_ADDR`. `MI32_DWR` data can be byte-enabled by the `MI32_BE` vector. The firmware logic acknowledges the write

transaction by asserting the MI32_ARDY signal when it has processed the Write request. A standard write transaction timing diagram is shown in the Figure 6.3.

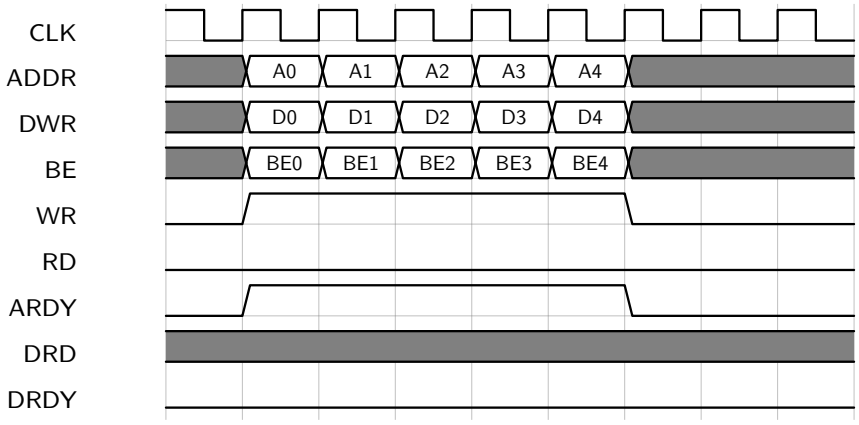

Figure 6.3: *MI32 standard write transaction*

The firmware logic is able to delay the write transaction if necessary. This is done by deasserting the MI32_ARDY signal until the logic is able to process the Write request. Until the MI32_ARDY signal is asserted, the write transaction is suspended and MI32_ADDR, MI32_DWR and MI32_BE values do not change. A delayed write transaction timing diagram is shown in the Figure 6.4.

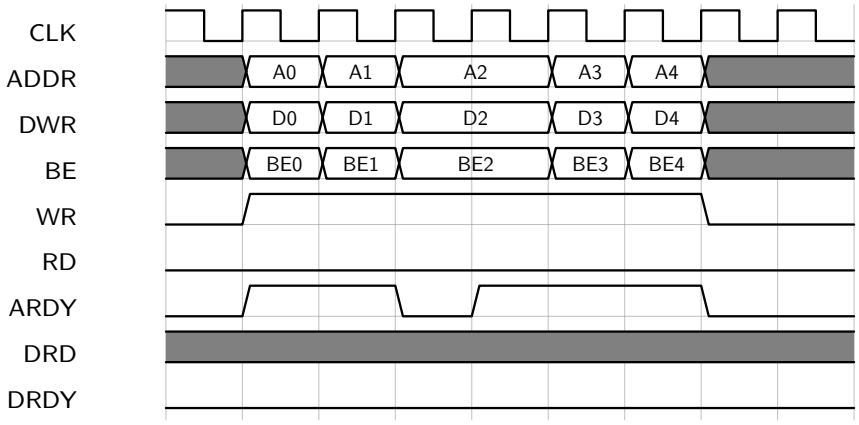

Figure 6.4: *MI32 delayed write transaction*

6.3.3 Read transaction

A read transaction is initiated by assertion of the MI32_RD signal representing a Read request. During the read transaction, the target address is determined by MI32_ADDR. The firmware logic acknowledges the read transaction by asserting the MI32_ARDY signal when it has processed the Read request. As soon as the data are ready, they are assigned to MI32_DRD port and the MI32_DRDY signal is asserted. A standard read transaction timing diagram is shown in the Figure 6.5.

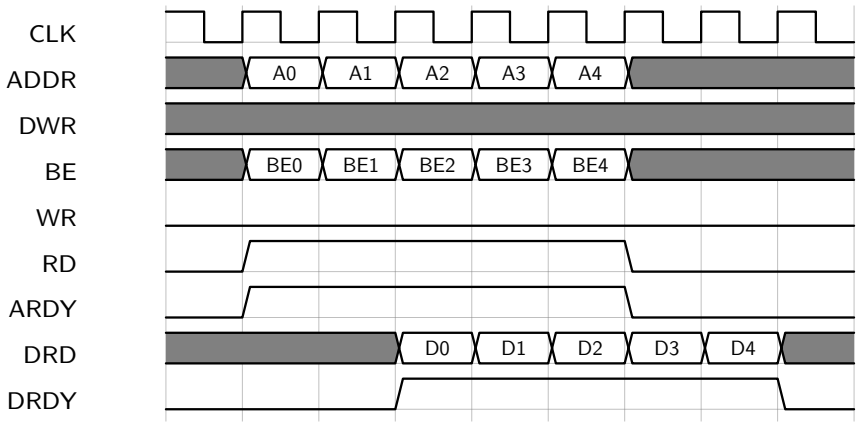

Figure 6.5: *MI32 standard read transaction*

The firmware logic is able to delay the read transaction if necessary. This is done by deasserting the `MI32_ARDY` signal until the logic is able to process the Read request. Until the `MI32_ARDY` signal is asserted, the read transaction is suspended and `MI32_ADDR` value does not change.

If the firmware logic is not able to read from the target address immediately, it can deassert the `MI32_DRDY` signal and so delay the read transaction. Until the `MI32_DRDY` signal is asserted, the read transaction is suspended. A delayed read transaction timing diagram is shown in the Figure 6.6.

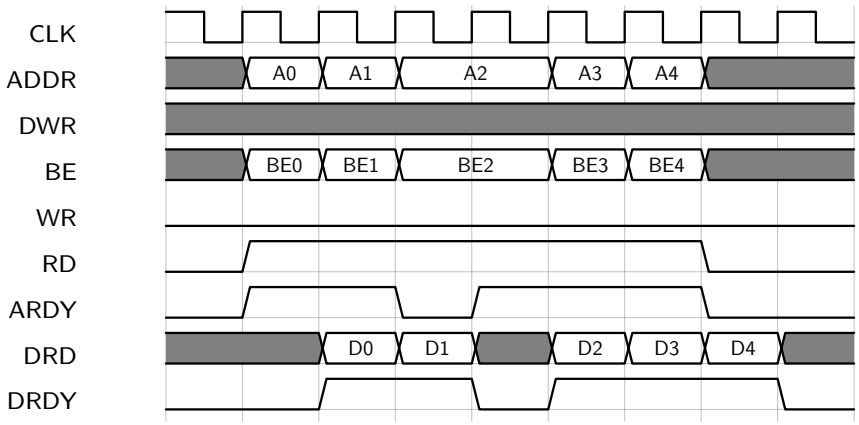

Figure 6.6: *MI32 delayed read transaction*

7 Simulation and Testing

Netcope offers P4 Lab, where it is possible to test the generated firmware running in the NFB-100G2Q card. The description of the lab is out of scope of this document, please refer to Help section of the Netcope P4 Cloud web interface.

Since P4 synthesis may take long time, it may be beneficial to simulate your P4 code in advance. We recommend using some of the freely available tools. A comprehensive tutorial can be found at Github

Revisions

Version	Date	Description
2.1	08/2018	Updated features list, updated resource consumption.
2.0	06/2018	Added description of cuckoo hashing, updated know limitations, added description of packet multiplication.
1.2	03/2018	Support for bit operations and Netcope P4 core identification
1.1	01/2018	Support for computing hash functions, added netlist guide, general fixes
1.0	10/2017	Initial revision

Questions

Please contact us for additional information about the Netcope P4 product.

Ordering Information

Please contact Netcope Technologies for pricing and additional information about the product purchasing.

Contact Information

Netcope Technologies, a.s.
Sochorova 3232/34, Brno 61600
Czech Republic, EU

Web: www.netcope.com
Email: info@netcope.com
Phone: +420 530 510 670

Confidential

Do not distribute this material without written approval from Netcope Technologies, a.s.

Disclaimer