Politecnico di Torino

POLITECNICO DI TORINO

Master of Science in Electronic Engineering
Graduation Session of March/April 2022

Design, Verification and Integration of a hardware accelerator for image composition through alpha blending

Supervisor:
Prof. Maurizio Martina

Candidate:
Gabriele Perrone

Academic Year 2021/2022

Abstract

The goal of this work is to present the development phases of a hardware accelerator for image processing. This accelerator has been created to implement the alpha blending technique, thanks to which it is possible to superimpose two images using the transparency channel called alpha.

The accelerator was thought to extend the functionalities of a DMA controller that can exploit it to perform simple elaborations while moving data.

The work will focus on the logical design phase, where the accelerator will be analyzed in terms of internal structure. The correct functioning of the accelerator has been tested using a UVM environment to prove that the design specifications have been met. On a second time, it has been tested using a system-level environment to perform actual tests written in C language that involve authentic images.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hardware accelerator is a general term to indicate a category of hardware components found in computing systems that perform specific tasks to ease the main elaboration unit workload. A famous example of a hardware accelerator is the Digital Signal Processor (DSP), taking care of the data elaboration coming from analog sensors.

This work aims to present a hardware accelerator for image processing. The innovation of this accelerator is that it is not accessed directly by the system's main CPU. However, it will be connected to a DMA controller to extend its functionalities. So the main CPU can just instruct the DMA controller to move data and it will also obtain the image processing operation performed by the accelerator.

The document is organized as follows: in this chapter, it will be first described the task performed by the accelerator, then it will be illustrated an example of a typical system where it operates. In the second chapter, it will be discussed the logical design of the accelerator. The third chapter will present the unit-level verification phase of the accelerator using UVM. In the fourth chapter, the accelerator will be tested in an example environment using high-level tests. The fifth chapter will conclude this document by commenting on the achieved results.

1.1 Accelerator Functionality

1.1.1 Alpha blending

Alpha blending is the technique used in image processing for combining two or more pictures using the alpha channel contained in each pixel of the images. The alpha channel is additional information contained in the pixel that indicates its level of transparency. Usually, it is given in a range from 0.0 to 1.0 where 0.0 indicates that a pixel is completely transparent while 1.0 indicates a pixel completely opaque.

Porter and Duff [1] described a complete algebra for digital image composition using the alpha channel. Let us now focus on the A over B operation they described, which is the most common operation that can be performed with two images. This operation permits to combine two images, A and B, so that A appears in the foreground and B appears in the background and can be summarized as follows: let C_A and C_B be two colors

components, regarding the same color, of two pixels in image A and B and let α_A and α_B be the alpha components, it is possible to calculate the result relative color component applying the equations:

$$\alpha_O = \alpha_A + \alpha_B(1 - \alpha_A) \tag{1.1}$$

$$C_O = \frac{C_A\alpha_A + C_B\alpha_B(1 - \alpha_A)}{\alpha_O} \tag{1.2}$$

where α_O and C_O represent the alpha and the color component of the pixel in the resulting image.

In the specifications document of the accelerator, there was written not to consider the alpha channel of image B and to fix the alpha channel of the resulting image to 1.0. This can be achieved easily by substituting $\alpha_B = 1$ in Equation 1.1:

$$\alpha_O = \alpha_A + (1 - \alpha_A) = \alpha_A + 1 - \alpha_A = 1$$

Applying the obtained result to Equation 1.2 the final formula required by the accelerator is:

$$C_O = C_A\alpha_A + C_B(1 - \alpha_A) \tag{1.3}$$

The color components in the resulting image's pixels will be a linear combination of the color in images A and B weighted by the alpha channel of image A.

1.1.2 Task performed by the accelerator

The task performed by the accelerator was derived directly from its specification document. According to it, the accelerator should receive two images from two different interfaces, one for foreground and one for background and perform the composition A over B using Equation 1.3. As previously reported, the alpha channel of the background (image B) was not to be considered. The alpha channel for each pixel of the foreground (image A) should be sent to the accelerator separately using a third interface. The accelerator should use a fourth interface for sending out the produced result image. The specification documents also stated that all the images, foreground, background and result, should have the same number of pixels and aspect ratio.

Let us see a graphical example. In Figure 1.1 is reported the foreground image to combine with the background image reported in Figure 1.2. The aspect ratio of the foreground image has been modified to meet the one of the background image by adding some white fills. Those represent the first two inputs of the accelerator. In Figure 1.3 is reported the third input, the alpha channel for the foreground image. Here, it is represented as a grayscale image where white is equal to an alpha of 1.0 while black is equal to an alpha of 0.0. In Figure 1.4 is reported the output produced by the accelerator. Notice how the car has been cut from the foreground image and is now on the top of the background image, merging with it alongside the edges.

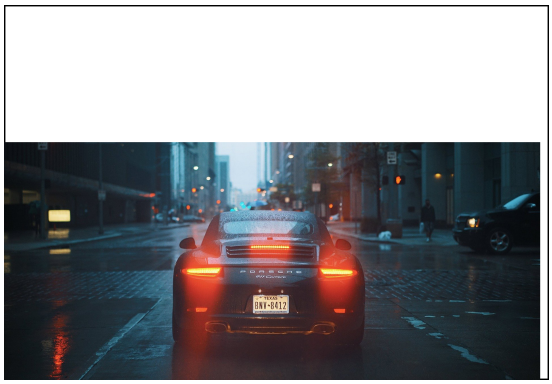

Figure 1.1: Foreground example image

Figure 1.2: Background example image

Figure 1.3: Alpha channel of the foreground image

Figure 1.4: Result example image

1.2 System Overview

In modern computing systems, data moving among main memory and peripherals is mainly done using Direct Memory Access (DMA). In this way, the CPU performs data movements indirectly by programming a DMA controller with the proper addresses and data size to move. In the meantime, the CPU can perform other operations. When the data movement is finished, the DMA will inform the CPU in the way it was programmed, for example, by an interrupt signal.

In the system presented here, the DMA controller not only can move data but, thanks to some hardware accelerators attached to it, this DMA controller can also perform some simple data elaboration. The hardware accelerator that was introduced previously will work alongside the DMA controller exchanging data directly with it. In Figure 1.5 is depicted an example system. Here it is possible to notice the position of the hardware accelerator (Stream Engine[1]).

Let us now focus on the hardware accelerator and its connections with the DMA controller. As stated previously, it should have four interfaces for data exchange with

[1]The name Stream Engine comes from the naming convention used inside the project where both the DMA controller and the hardware accelerator were developed, it will be clarified later in the text.

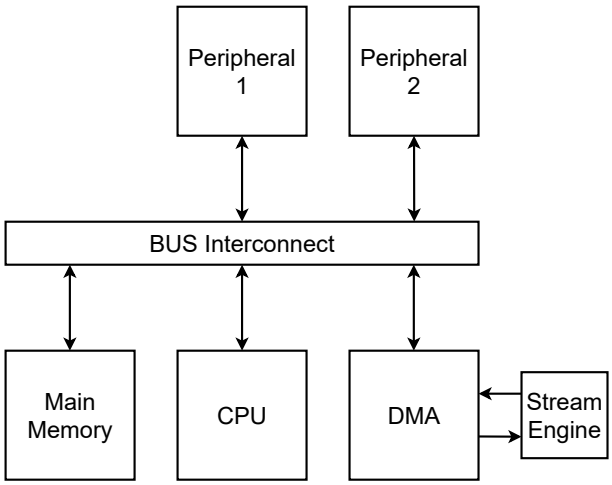

Figure 1.5: Example system where the hardware accelerator (Stream Engine) operates

the DMA. The interfaces for receiving the two images and the alpha data will use the AMBA®4 AXI4-Stream[2] protocol in the receiving mode. The interface for sending out the produced data will use the same protocol in transmitting mode. Since all the data come from the DMA controller, it will have the counterparts of those four interfaces. In particular, the controller is divided into channels. Each of them has one AXI4-Stream interface for transmitting and one for receiving. That means that the accelerator will be connected to at least three different channels: the first will send one of the three data streams and receive the result image data stream. The other two will just send the remaining data streams.

Since the specification document stated that the color format of the foreground, background, and result images should be configurable and can differ from each other, three groups of General Purpose Inputs (GPI) were added to the accelerator. Those inputs are connected to the DMA controller General Purpose Outputs (GPO) and are set by it before sending the actual input images.

It is now clear that the name Stream Engine was given to the accelerator because its communication interfaces rely on the AXI4-Stream protocol. The name Stream Engine will be used to refer to the hardware accelerator from now on.

In Figure 1.6 is possible to observe an example connection between the Stream Engine and the DMA controller. In this case, channel 0 will send the foreground image and receive the resulting image. Channel 1 will send the background image and channel 2 the alpha data stream. In this case, the color formats are handled by the channels that handle the respective image, channel 0 the foreground and the result, channel 1 the background. This is not mandatory but highly recommended since it gives coherency between data and data format to who will program the software for the system. In particular, when the DMA controller is programmed to use the Stream Engine, the color format of one image and the image itself will belong to the same channel.

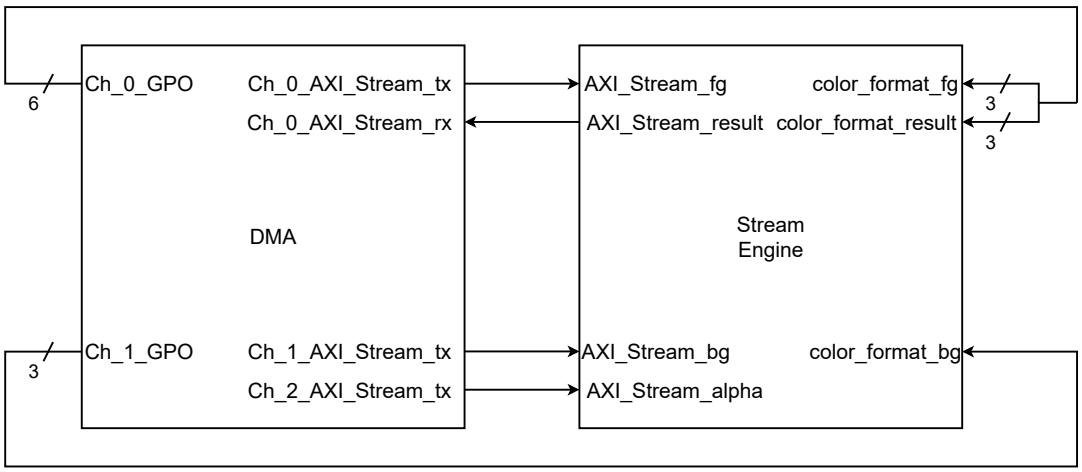

Figure 1.6: Example connection between the DMA controller and the hardware accelerator (Stream Engine)

Chapter 2

Design

In this chapter, it will be presented the logical design of the Stream Engine. First, it will be introduced the adopted design strategy. Then the chapter will continue with the design analysis of the blocks that compose the Engine.

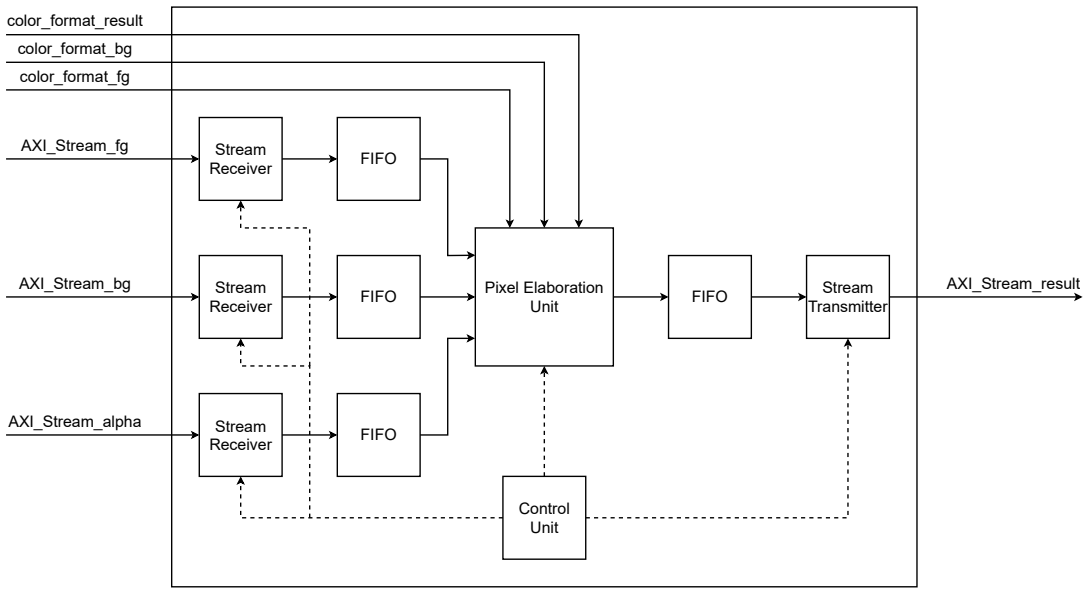

Figure 2.1: Stream Engine internal organization

The idea beyond the design was to divide the Engine into small blocks. Each block was developed to handle a specific task inside the Engine. Each block contains an FSM and a specific datapath to handle that particular task. Recalling that the Engine has three AXI4-Stream interfaces for receiving data, the `stream_receiver` block was designed to handle one interface, independently of the nature of the data. Each `stream_receiver` is connected to a FIFO that stores the incoming data. Connected to those FIFOs, there is the heart of the Engine, the `pixel_elaboration_unit`. This block retrieves pixels and alpha data from the three FIFOs to produce the result pixels. Those pixels are stored in another FIFO. The content of this last FIFO is pulled from the last block in the data flow, the `stream_transmitter`. It handles the AXI4-Stream output interface sending data

out. Finally, there is the `control_unit` block which task is to make the other blocks work correctly together.

Figure 2.1 clarifies the Engine internal organization by showing the blocks introduced above.

2.1 FIFO

When the design of the Stream Engine started, this component was already available inside the project where both the DMA controller and the Engine were developed. It has been reused as it was since its functionalities matched precisely the Engine's needs. All the other blocks were shaped almost independently from each other around this one because, as illustrated in Figure 2.1 all the used FIFOs create memory barriers between the other blocks. Since it was not designed as a part of the Engine work, its internal structure will not be detailed. Only its main functionalities and the related interfaces will be described.

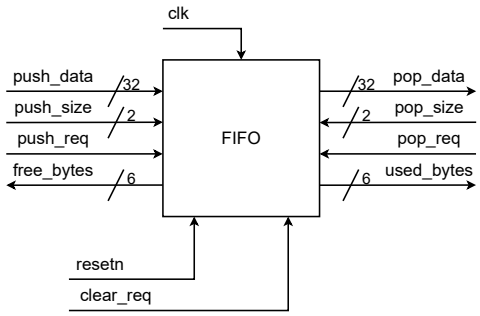

Figure 2.2: FIFO block

From an external point of view, the FIFO can be represented as reported in Figure 2.2. It has two main interfaces, one for pushing data into it and one for popping data. The pushing interface is formed by all the signals which name starts with `push_` plus the `free_bytes` signal. Its functioning is quite simple: at the rising edge of the `clk` signal if `push_req` is asserted the number of bytes indicated by `push_size` are taken from the lower byte lanes of `push_data` and stored in the FIFO. The `free_bytes` signal indicates at any time how many bytes the FIFO can accept without taking into consideration an incoming push request. The popping interface instead is formed by all the signals which name starts with `pop_` plus the `used_bytes` signal. Also, its functioning is quite simple: at the rising edge of the `clk` signal, if `pop_req` is asserted, the number of bytes indicated by `pop_size` are taken from the FIFO and put in the lower byte lanes of `pop_data`. The `used_bytes` signal indicates at any time how many bytes can be popped from the FIFO without taking into consideration an incoming pop request. The last remaining signal, `clear_req`, can be used to clear the content of the FIFO synchronously.

The FIFO can be configured to match the design needs. In particular, the data width in number of bits and the memory depth can be chosen freely. To easily represent the Engine and producing all the figures contained in this chapter, those values were set to: `data_width` = 32, and `fifo_depth` = 8. In the Engine, the data width of the FIFO always

matches the AXI4-Stream data width that should match the DMA controller data width, which can assume only the values 32, 64, 128 or 256 bits. The FIFO depth instead is a configuration parameter of the Engine. It can assume all the powers of 2 values. Together with the data width parameter, it set the number of bytes the FIFO can contain. An analysis of a good choice of this last parameter will be given during the design description of the other blocks.

2.2 Stream receiver

The stream receiver is the block that handles one input interface of the Stream Engine and a FIFO to store the incoming data. There are three instances of this block in the Engine, one for each input interface. From the outside, it looks like as reported in Figure 2.3.

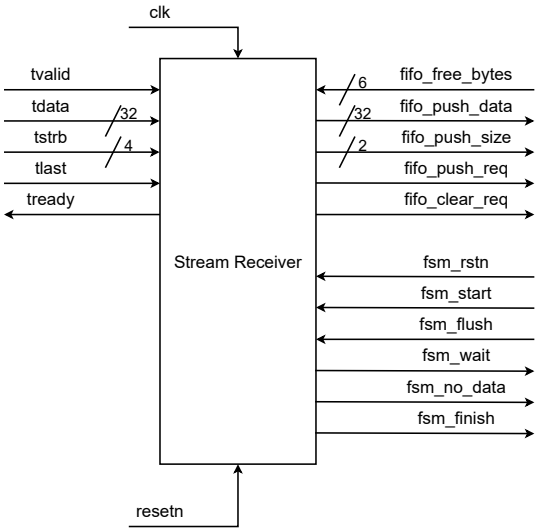

Figure 2.3: Stream receiver block

Its design started with the analysis of the AXI4-Stream protocol and, in particular, the version adopted by the DMA Controller. Using the terminology reported in the protocol specification, within this implementation, each channel of the DMA can send only data bytes and position bytes. Data bytes contain actual valid data, while position bytes are used only to fill gaps between data bytes. Furthermore, position bytes can be sent by the DMA controller only during the last transfer and to fill it up to the bus size. The subset of signal used for the protocol is reported in Table 2.1.

The protocol functioning can be summarized as follows: when a rising edge of the clock occurs, and both **TVALID** and **TREADY** are asserted, valid data must be present on **TDATA** and the receiver must accept it on the same edge. This is called a transfer. The "n" present in the signals **TDATA** and **TSTRB** indicates the number of byte lanes. The transmitter can signal to the receiver which lanes contain position bytes by deasserting during the transfer the corresponding **TSTRB** bit. A set of transfers is called a packet. Concerning the Stream Engine, a packet from the DMA contains an entire image or the

Table 2.1: Subset signal list for the adopted version of the AXI4-Stream protocol

Signal	Source
ACLK	Clock source
ARESETn	Reset source
TVALID	DMA
TREADY	Engine
TDATA[(8n-1):0]	DMA
TSTRB[(n-1):0]	DMA
TLAST	DMA

alpha data for an image. The transmitter also asserts **TLAST** during the last transfer to signal the end of the packet. Other protocol details will be highlighted during the analysis of the stream receiver block.

The design continued by splitting the block into a small datapath and an FSM to control it. The datapath was necessary to handle some operations required to link the AXI4-Stream interface with the FIFO. Both those parts will be analyzed separately in the following subsections.

2.2.1 Datapath

The datapath of the stream receiver is quite simple. Its RTL representation[1] is reported in Figure 2.4. In the upper part, first, the `TSTRB` signal is converted in a number of bytes, then the result is converted again to match the `fifo_push_size` encoding where the actual number of bytes is decreased by one. The `TSTRB` conversion is done to tell the FIFO how many bites are going to be stored. In this case, it is sufficient to use a 1' counter because the valid data are always packed in the lower byte lanes. The two registers on `new_bytes` and `tdata` have been inserted to perform easier back to back reception.

2.2.2 FSM

The design of the stream receiver FSM started by analyzing some possible situation that can happen between the AXI4-Stream interface and the FIFO. To understand them better, they were plotted using waveform representation. The most important analyzed situations are reported in Figure 2.5 and in Figure 2.6.

In Figure 2.5 it is possible to see first the FSM in `RESET` state (cycle 1), then the reset signal is deasserted and the FSM goes to `IDLE` state (cycle 2). The start signal is then asserted, and the FSM goes to the `READY` state where it waits for new transfers from the DMA by asserting the `TREADY` signal (cycle 3). In cycle 4, the first transfer is sent from the DMA, and the FSM moves to the `PUSH` state and stores it in the FIFO (cycle 5). In the same state, it can keep accepting new transfers. In cycle 6, the DMA is not sending a new transfer for the moment, and the FSM moves to the `WAIT_DATA` state (cycle 7), where

[1] In all the reported RTL representation, a signal in bold indicates an input while a signal underlined indicates an output.

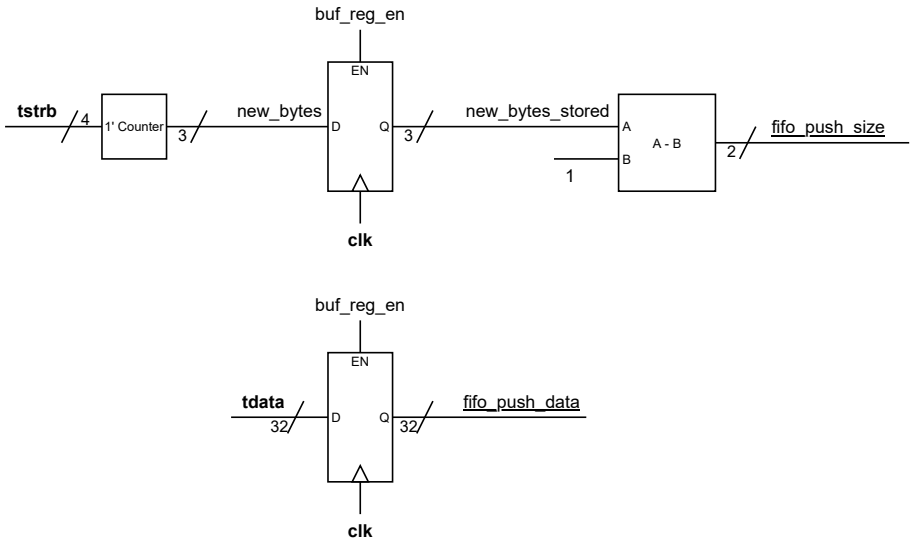

Figure 2.4: Stream receiver datapath

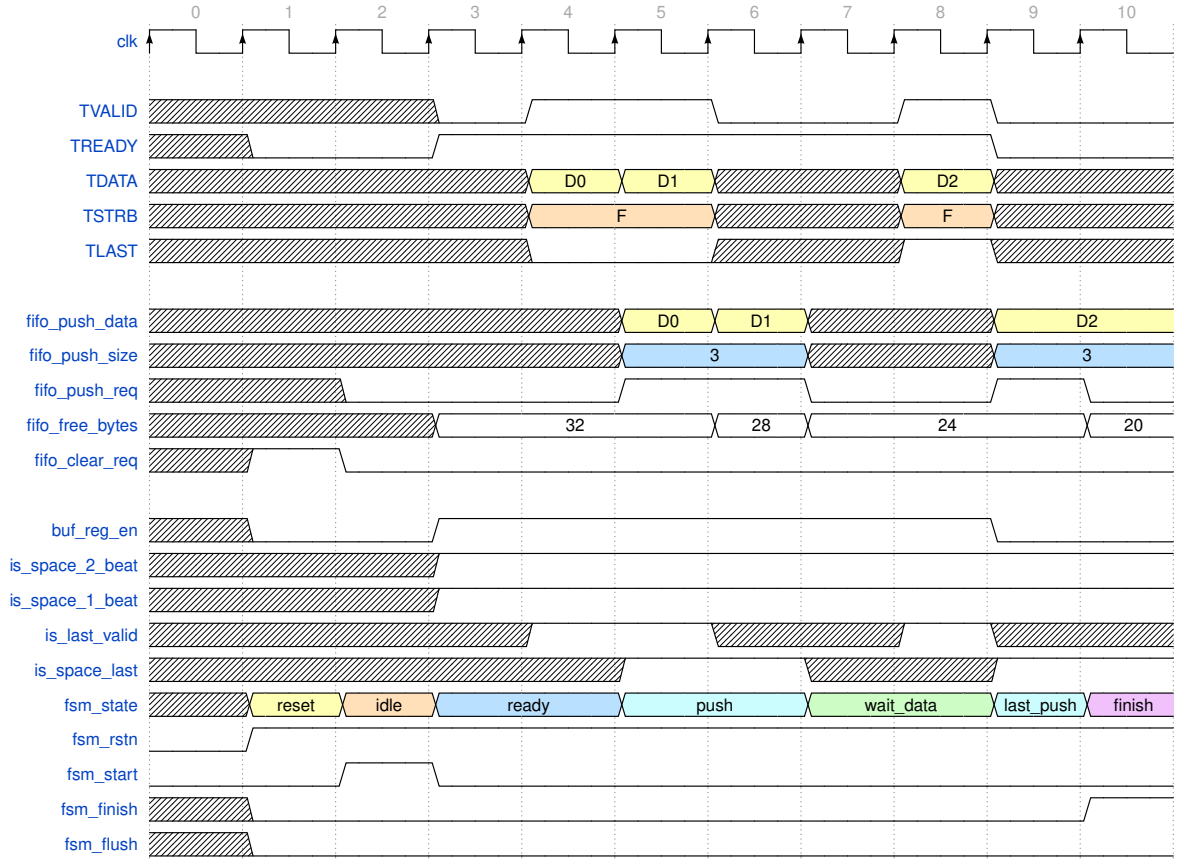

Figure 2.5: Stream receiver, waveform for FSM design - part 1

it keeps the `TREADY` signal asserted waiting for the DMA to start sending transfers again. In cycle 8, the last transfer of the packet is sent by the DMA, and the FSM moves to

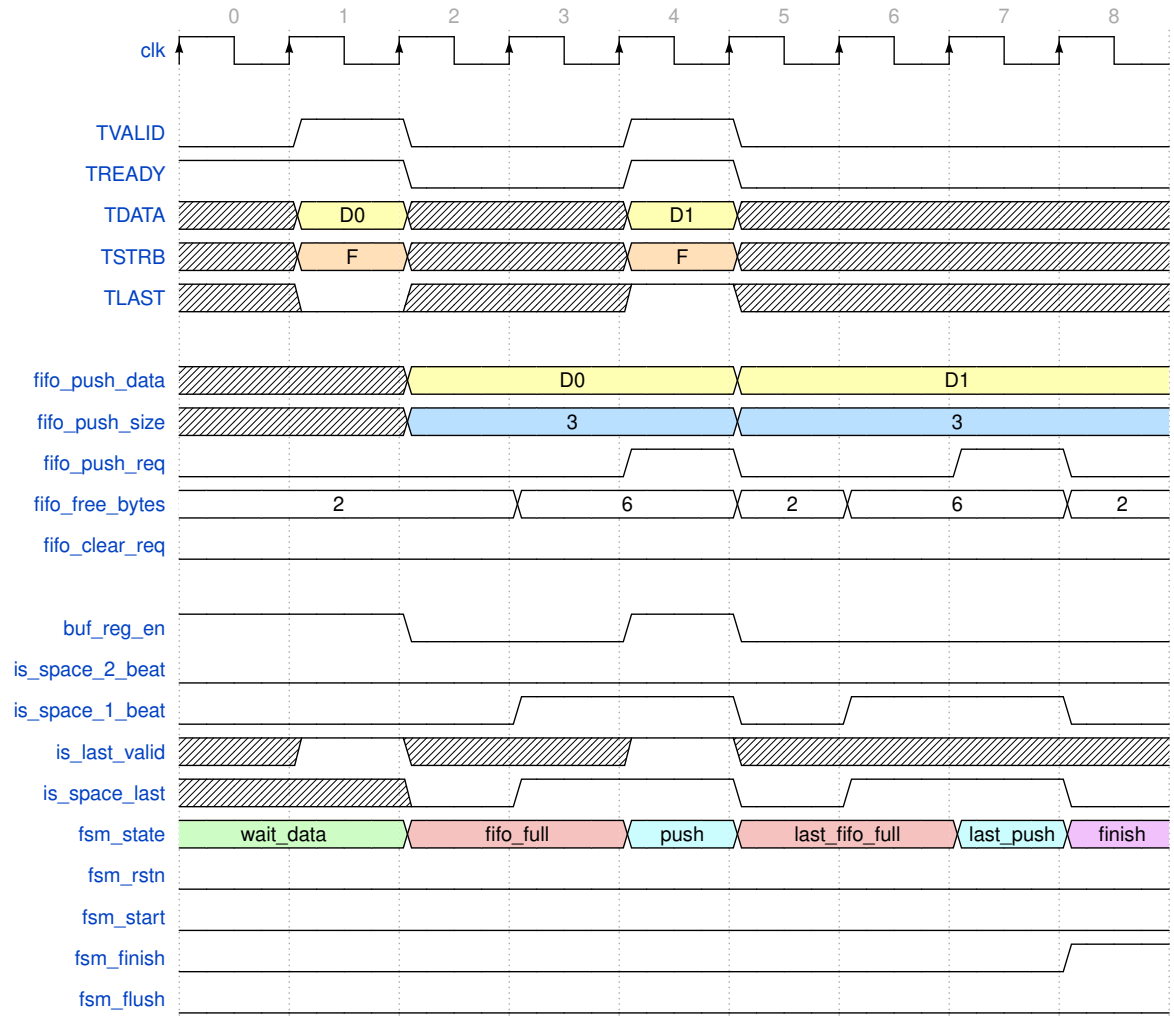

Figure 2.6: Stream receiver, waveform for FSM design - part 2

the `LAST_PUSH` state where the last transfer is stored. Finally, the FSM moves directly to the `FINISH` state, where it signals to the control unit block that its packet is fully arrived (cycle 9).

In Figure 2.6 are reported other typical situations. In cycle 1, the FSM is in the `WAIT_DATA` state, and a new transfer is coming from the DMA. Since there is no space in the FIFO to store it, the FSM moves to the `FIFO_FULL` state, waiting for some space in the FIFO. In cycle 3, the FIFO now has enough space to store the transfer, so the FSM moves to the `PUSH` state in cycle 4. In the same cycle, a new transfer, the last one of the packet, is coming from the DMA, but there is not enough space to store it. The FSM goes to the `LAST_FIFO_FULL` state after having stored the previous transaction waiting for space in the FIFO for the last transfer (cycle 5). In cycle 6, there is enough space to store the last transfer, so the FSM moves to the `LAST_PUSH` state.

From those waveforms, it was derived an initial FSM state diagram. After the integra-

tion of all the blocks, it was improved, arriving at a final version reported in Figure 2.7[2]. When the asynchronous reset is asserted, the FSM goes and remains in the `RESET` state. In this state the FSM clears its corresponding FIFO by asserting `fifo_clear_req`. After the Stream Engine is released from asynchronous reset, the FSM moves to the `IDLE` state asserting `fsm_no_data` to inform the control unit block. The FSM is started by the signal `fsm_start` coming from the control unit block. It then enters the `READY` state waiting for the DMA to start sending data. In this state, the FSM keeps `fsm_no_data` asserted to inform the control unit block that the packet has not yet started. It also asserts `tready` to inform the DMA's channel that this stream receiver is ready to accept the packet, and `buf_reg_en` because valid data arrive on the same cycle of `tvalid` signal assertion. Those three states are highlighted in green in the figure and represent the initial phase of the FSM.

During the reception of a complete packet from the channel, the stream receiver FSM spends most of the time in the states `PUSH`, `WAIT_DATA` and `FIFO_FULL`. Those three states, highlighted in blue in the figure, represent the main phase of the FSM. When the FSM remains in the `PUSH` state, it is accepting a new transfer from the AXI4-Stream interface and storing the old one in the FIFO. To do that, the FSM asserts `tready`, `buf_reg_en` and `fifo_push_req`. For the FSM to remain in this state, the FIFO must have enough free bytes to store a number of bytes equal to twice the bus size. The reason is that the FIFO does not consider an incoming push request in calculating the number of free bytes. When the DMA has no new transfers to send but has not finished the packet, it pulls down the `tvalid` signal, and the FSM goes to `WAIT_DATA` state. In this state the FSM keeps `tready` and `buf_reg_en` asserted to be ready for data reception. When the DMA starts again sending new transfers, the FSM leaves this state. In case the DMA is sending a transfer, but there is no space in the FIFO, the FSM can go to the `FIFO_FULL` state. In this state, the `tready` signal is pulled down. This prevents the DMA from completing a new transfer. The FSM instead asserts the `fsm_wait` signal to inform the control unit that data reception cannot continue because the FIFO is full. The FSM stays in this state until there is enough space in the FIFO to store the content of the bus.

When the FSM detects the last transfer, it moves to a set of states to handle the end of the packet, `LAST_PUSH`, `LAST_FIFO_FULL` and `FINISH`. Those states are highlighted in orange in the figure. `LAST_PUSH` is entered if, on the detection of the last transfer, there is enough space in the FIFO to store it, `LAST_FIFO_FULL` otherwise. When the FSM is in `LAST_FIFO_FULL` state, it asserts the `fsm_wait` signal to tell the control unit that the packet reception is blocked because there is not enough space in the FIFO. When there is enough space, the FSM moves to `LAST_PUSH` and stores the last transfer by asserting `fifo_push_req`. From `LAST_PUSH`, the FSM moves to `FINISH` where it asserts the `fsm_finish` signal to inform the control unit that the reception of the packet is completed.

The `FLUSH` state, highlighted in red in the figure, is entered when the control unit block asserts the `fsm_flush` signal and the FSM is in the `FIFO_FULL` state. It means something went wrong during the elaboration and the ongoing process needs to be aborted. In the `FLUSH` state, the FSM keeps only `tready` asserted to accept all the transfers without storing

[2]The convention used for all the FSM state diagrams regarding the output signals is that when they are present in a state, they are asserted (only in case of 1-bit signals) or accompanied with their value.

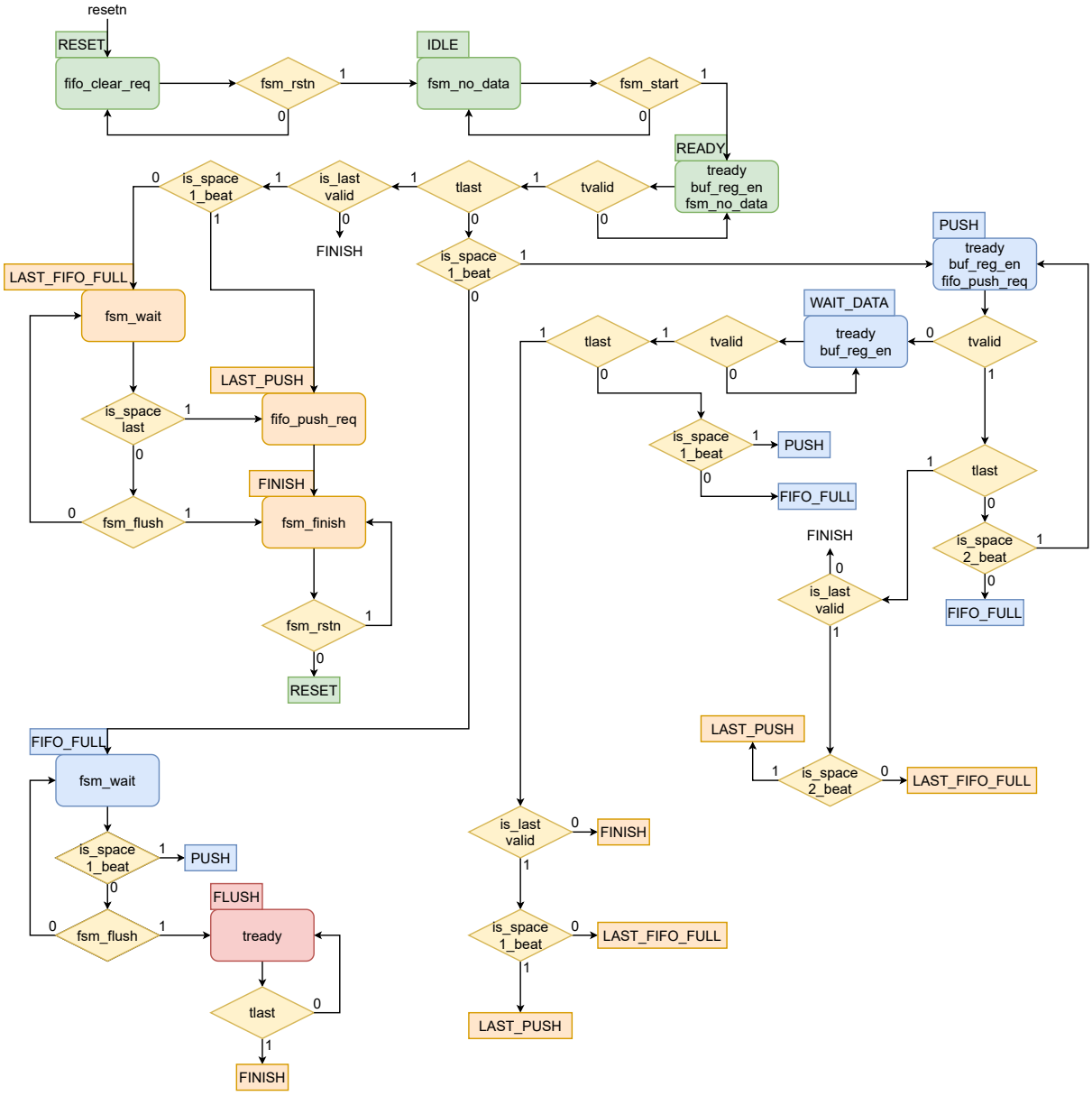

Figure 2.7: Stream receiver, FSM state diagram

them. This is done because the AXI4-Stream protocol must be terminated correctly at all interfaces. When the last transfer is detected, the FSM moves from the `FLUSH` state directly to the `FINISH` state.

The RTL representation of the condition signals used by the FSM is reported in Figure 2.8. The condition `is_space_last` is used to check if there is enough space in the FIFO for the last transfer. `Is_space_1_beat` is used to check if there is enough space in the FIFO for a number of bytes equal to the bus width, while `is_space_2_beat` is used to check if there is enough space for a number of bytes equal to twice the bus width. `Is_last_valid` is used to check if the last transfer contains at least one valid byte. The reason is that the DMA can send the last transfer made of only position bytes.

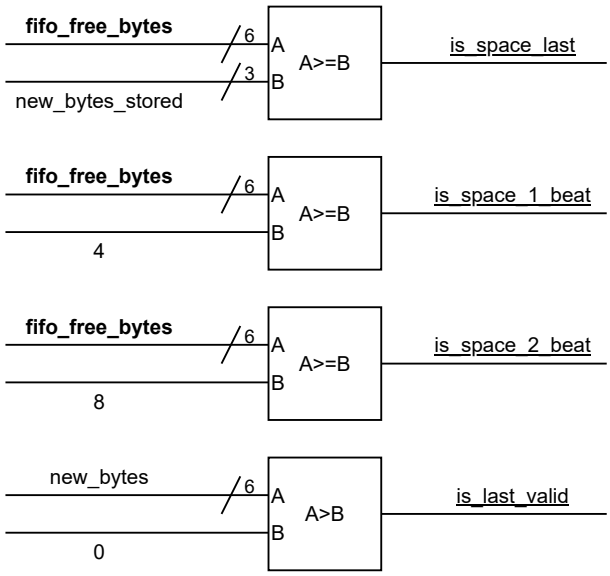

Figure 2.8: Stream receiver, FSM condition signal evaluation

2.3 Stream transmitter

The stream transmitter is the block that retrieves the result data from the result FIFO and sends them back to the DMA handling the Stream Engine output interface. From the outside, it looks like as reported in Figure 2.9.

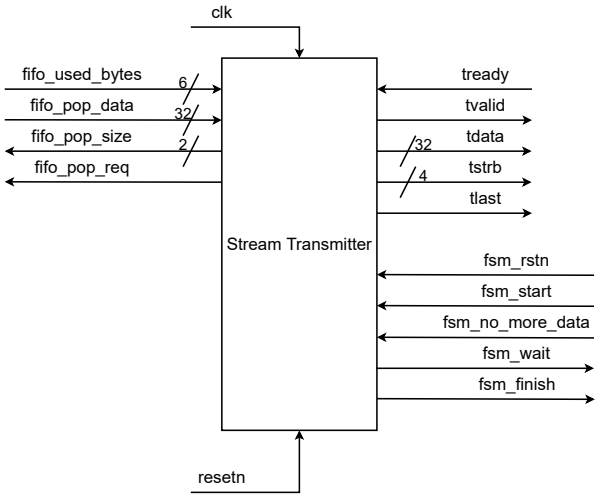

Figure 2.9: Stream transmitter block

The DMA adopts the exact version of the AXI4-Stream protocol both for transmitting and receiving data, so the stream transmitter here uses the same protocol used by the DMA to send data. Here, the source of the signals presented in Table 2.1 is inverted.

The same design approach used for the stream receiver is adopted here. The block has been split into a small datapath and an FSM to control it. Again the datapath was

necessary to perform some signal conversion required to link the FIFO to the AXI4-Stream interface. Both those parts will be analyzed separately in the following subsections.

2.3.1 Datapath

The RTL representation of the stream transmitter datapath is reported in Figure 2.10. In the upper part, first, the number of bytes to pop from the FIFO is selected by the FSM between the maximum and the actual remaining in the FIFO. This number on one side is converted to match the `fifo_pop_size` encoding[3], on the other side it is used to generate the `TSTRB` signal for the transfer. The two registers on `pop_bytes_conv` and `fifo_pop_data` have been inserted to perform easier back to back data transmission.

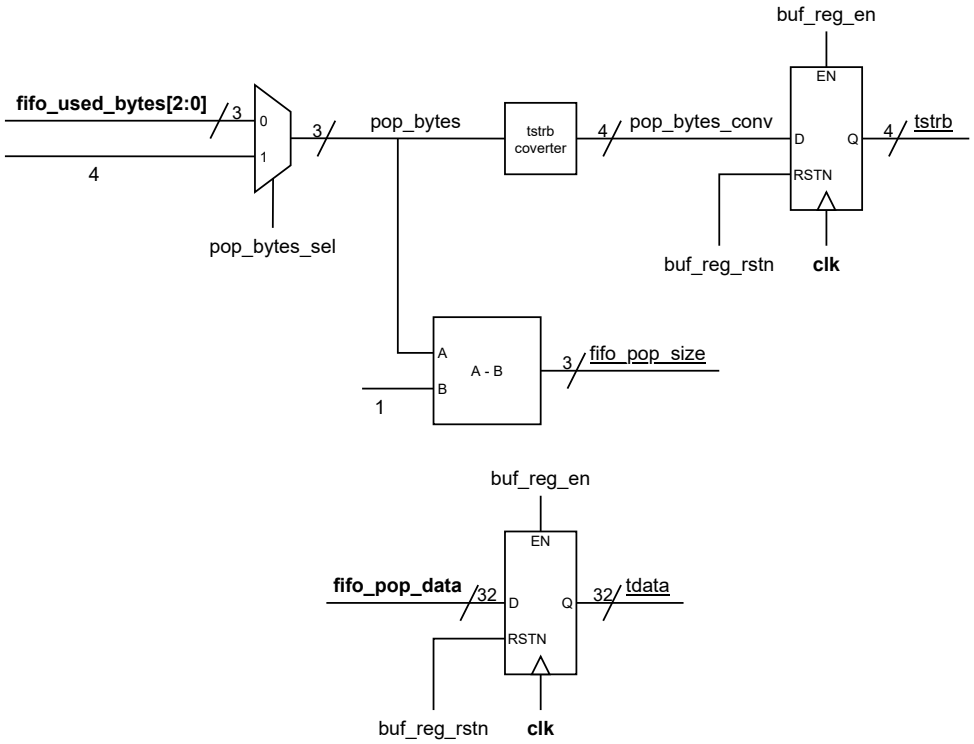

Figure 2.10: Stream transmitter datapath

2.3.2 FSM

The design of the stream transmitter FSM started by analyzing some possible situation that can happen between the FIFO and the AXI4-Stream interface. To understand them better, they were plotted using waveform representation. The most important analyzed situations are reported in Figure 2.11 and in Figure 2.12.

In Figure 2.11 the FSM is first in `RESET` state (cycle 1), in this state `TDATA` and `TSTRB` registers are reset to avoid data leakage on the bus. Then the reset signal is deasserted,

[3]It is the same encoding of the `fifo_push_size`.

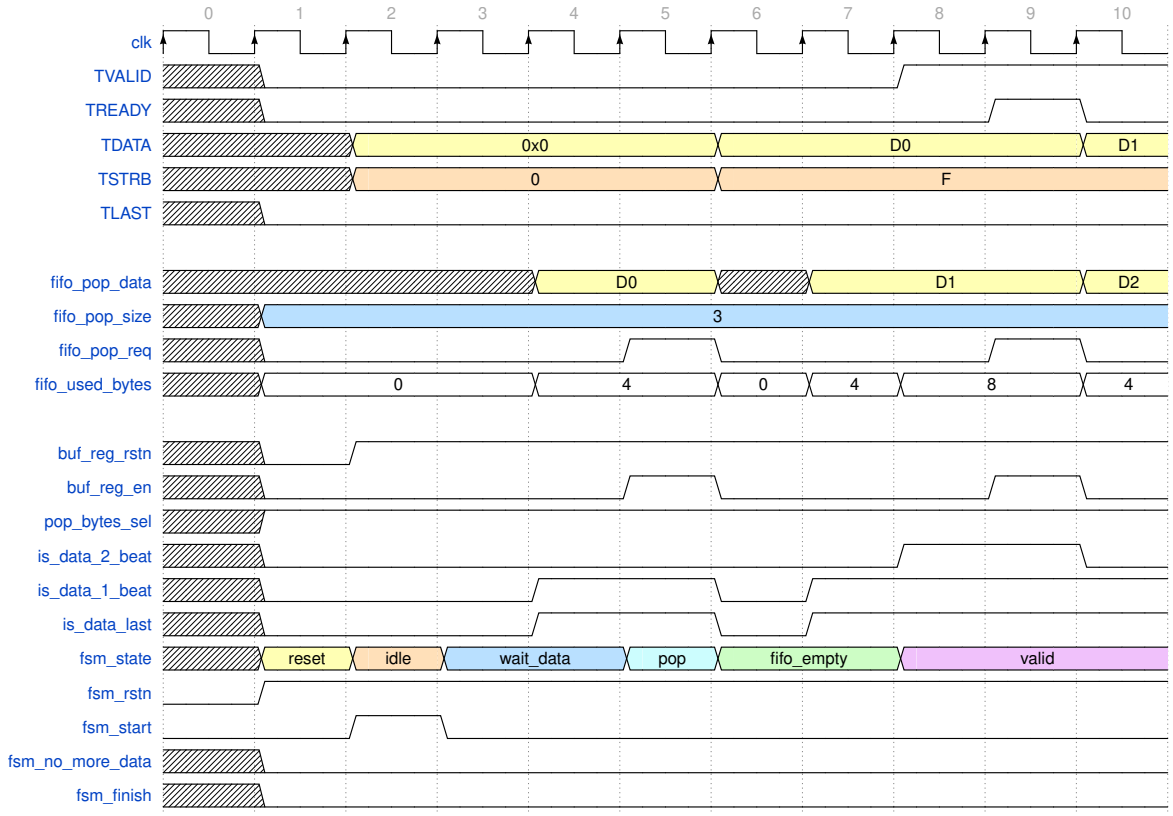

Figure 2.11: Stream transmitter, waveform for FSM design - part 1

and the FSM goes to the `IDLE` state (cycle 2). The start signal is then asserted, and the FSM goes to the `WAIT_DATA` in cycle 3. Since the FIFO does not have enough bytes to be sent in this cycle, the FSM remains in this state for the next cycle. In cycle 4, the FIFO now has enough data, and the FSM moves to the `POP` state where the FSM retrieves the first data (cycle 5). The FSM moves then to the `FIFO_EMPTY` state because the FIFO does not contain enough data for sending two transfers (cycle 6). In cycle 7, the FIFO has enough data for one transfer again, so the FSM moves to the `VALID` state (cycle 8) where it asserts the `TVALID` signal. The FSM remains in this state until the DMA can accept the transfer like in cycle 9. In this cycle, it is also possible to observe that the FSM retrieves new data from the FIFO only if the previous transfer completes. The FSM remains in the `VALID` state also in cycle 10 because the FIFO had enough data for two transfers in the previous cycle.

In Figure 2.12 are reported other typical situations. In cycle 1, the FSM is in the `VALID` state, and the DMA is accepting a transfer. In the same cycle, the FIFO does not have enough data for two transfers, so the FSM moves to the `FIFO_EMPTY` state (cycle 2). The control unit of the Stream Engine has previously asserted the `fsm_no_more_data` signal to inform the FSM that the FIFO is not going to receive any more data. Because of that and because `TDATA` register contains valid data that the DMA has not accepted yet, the FSM moves to the `VALID_PRE_LAST` state (cycle 3). This state is similar to the `VALID` state. The difference is that in this state, the FSM prepares to retrieve the last data from the FIFO

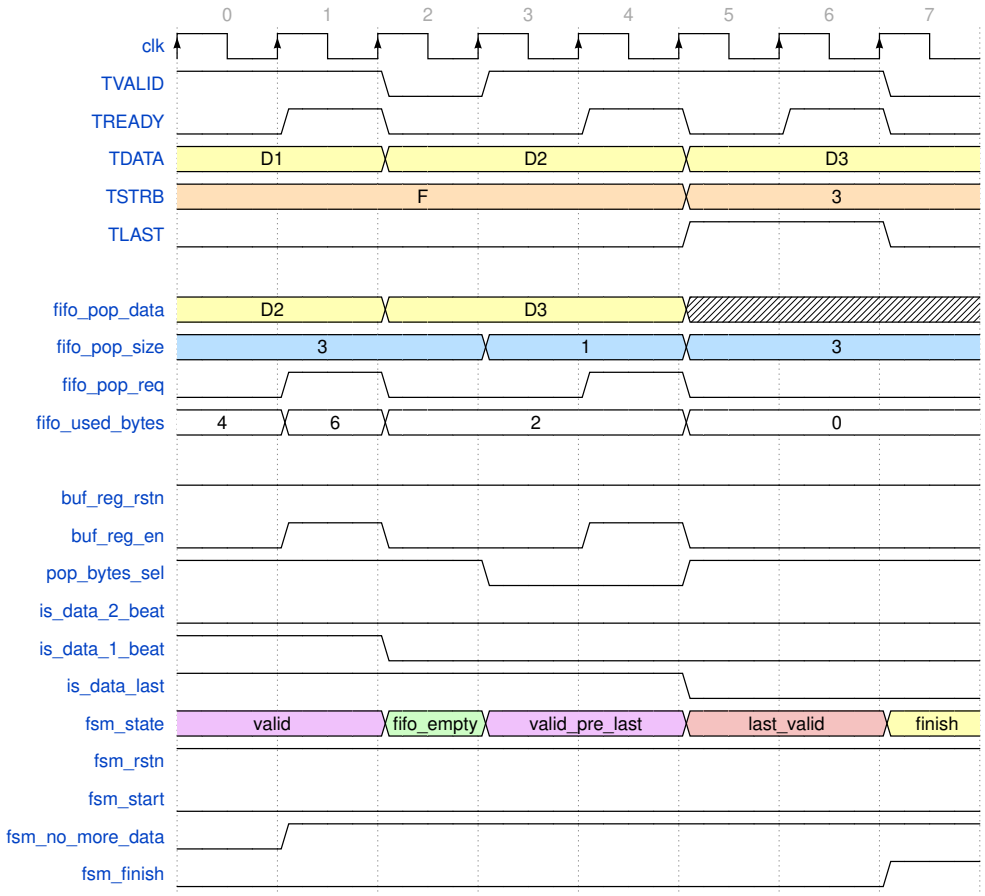

Figure 2.12: Stream transmitter, waveform for FSM design - part 2

by changing the `pop_bytes_sel` signal. In cycle 4, the DMA accepts the transfer, and the FSM prepares the last transfer. So the FSM moves to the `LAST_VALID` state and asserts the `TLAST` signal waiting for the DMA to accept the last transfer (cycle 5). In cycle 6, the DMA accepts the last transfer, and finally, the FSM moves to the `FINISH` state (cycle 7), where it signals the end of the packet to the control unit by asserting the `fsm_finish` signal.

From those waveforms, it was derived an initial FSM state diagram. After the integration of all the blocks, it was improved, arriving at a final version reported in Figure 2.13. When the asynchronous reset is asserted, the FSM goes and remains in the `RESET` state. In this state, the FSM clears its buffer register by asserting `buf_reg_resetn`. After the Stream Engine is released from asynchronous reset, the FSM moves to the `IDLE` state. The FSM is started by the signal `fsm_start` coming from the control unit block. It then enters the `WAIT_DATA` state, waiting for the presence of data in the FIFO. In this state, the FSM keeps `fsm_wait` asserted to inform the control unit block that the packet transmission is not yet started. Those three states are highlighted in green in the figure and represent the initial phase of the FSM.

During the transmission of a complete packet to the DMA, the stream transmitter FSM spends most of the time in the states `VALID` and `FIFO_EMPTY`, passing first from the

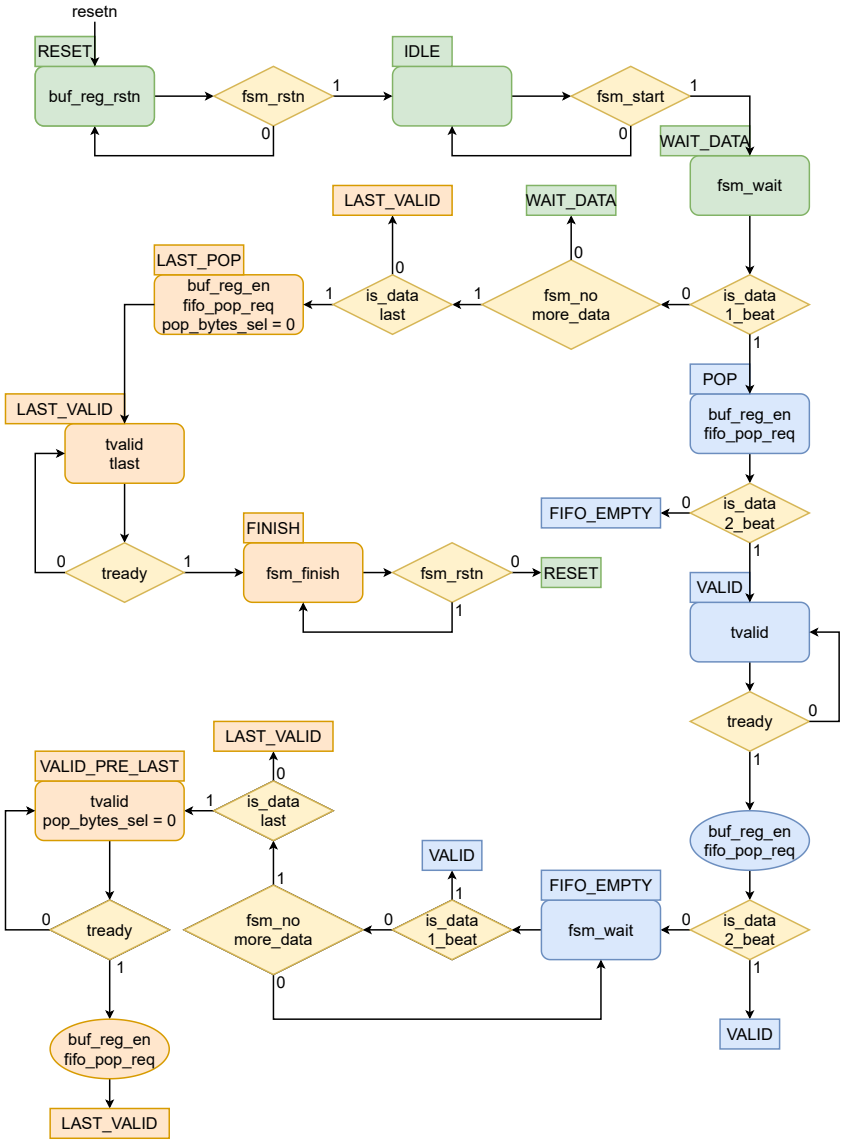

Figure 2.13: Stream transmitter, FSM state diagram

`POP` state. Those three states, highlighted in blue in the figure, represent the main phase of the FSM. When enough data for a transfer is present in the FIFO, the FSM moves from the `WAIT_DATA` state to the `POP` state where the FSM retrieves the first data by asserting `fifo_pop_req` and `buf_reg_en` to store it in the buffer register. If the FIFO has enough data for at least other two transfers, the FSM moves to the `VALID` state; otherwise, it moves to the `FIFO_EMPTY` state. When the FSM is in the `VALID` state, a transfer is present on the bus, so the `tvalid` signal is asserted. The FSM remains in this state if the DMA has not yet accepted the current transfer or, after having accepted it, if the FIFO has enough data for at least other two transfers. In this state there are two Mealy outputs: `buf_reg_en` and `fifo_pop_req`. Those two signals are asserted only in the cycles the DMA accepts a transfer, signalled by the assertion of `tready`. This is done to retrieve a new transfer

from the FIFO when the last one was accepted. In this way, the FSM can maximize the throughput by always having a new transfer to send remaining in the same state. When there is not enough data in the FIFO, the FSM moves to the `FIFO_EMPTY` state, where it waits for enough data in the FIFO. In this state, the FSM keeps `fsm_wait` asserted to inform the control unit block that the packet transmission is blocked because the FIFO is empty.

When the control unit tells that the FIFO will not receive new data by asserting the `fsm_no_more_data` signal and the FSM is in the `FIFO_EMPTY` state, the FSM moves to a set of states to handle the end of the packet, `VALID_PRE_LAST`, `LAST_VALID`, `LAST_POP` and `FINISH`. Those states are highlighted in orange in the figure. The `VALID_PRE_LAST` state is entered if the FIFO still contains some data to send; the `LAST_VALID` state is entered otherwise. The `VALID_PRE_LAST` state is quite similar to the `VALID` state. The only difference is that the `pop_bytes_sel` signal is changed to 0 so that the number of bytes is not the maximum but the actual remaining in the FIFO. When the DMA accepts the second-to-last transfer, the FSM pops the last transfer from the FIFO using the Mealy outputs, `buf_reg_en` and `fifo_pop_req`, and moves to the `LAST_VALID` state. In this state, the FSM asserts the `tlast` signal together with the `tvalid` one to inform the DMA that this is the last transfer. When the DMA accepts this last transfer, the FSM moves to the `FINISH` state where it asserts the `fsm_finish` signal to inform the control unit that the transmission of the packet is completed. The `LAST_POP` state is similar to the `POP` state and has been created to handle a particular compatibility situation where a packet is made of only one transfer, and that transfer cannot fill the bus size. In this case the FSM goes from `WAIT_DATA` to `LAST_POP`. In this state the FSM retrieve the incomplete unique transfer from the FIFO by asserting `fifo_pop_req` and `buf_reg_en` with `pop_bytes_sel` set to 0. Them FSM moves then to `LAST_VALID` to close the AXI4-Stream protocol.

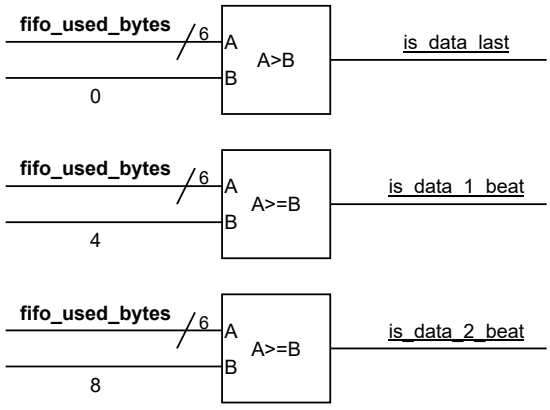

Figure 2.14: Stream transmitter, FSM condition signal evaluation

The RTL representation of the condition signals used by the FSM is reported in Figure 2.14. The condition `is_data_last` is used to check if the FIFO still contains some bytes for the last transfer. `Is_data_1_beat` is used to check if there is enough data in the FIFO to retrieve a complete transfer, while `is_data_2_beat` is used to check if there is enough data in the FIFO to retrieve two complete transfers.

2.4 Pixel elaboration unit

The pixel elaboration unit is the block that performs the actual pixel manipulations. Its design started with the analysis of the Engine requested functionalities. According to the design specification, it should apply the Equation 1.3 to all the incoming pixels taking into account the color format of the inputs and the output pixels. From the outside it appears like reported in Figure 2.15. The design approach was to divide the block in a fully combinatorial datapath and an FSM to control it. It was chosen to make the datapath combinatorial because the memory was already present in the FIFOs.

In the following subsections, first, an overview of the color formats used by the Engine will be given. Then the design of the datapath and the FSM will be analyzed separately.

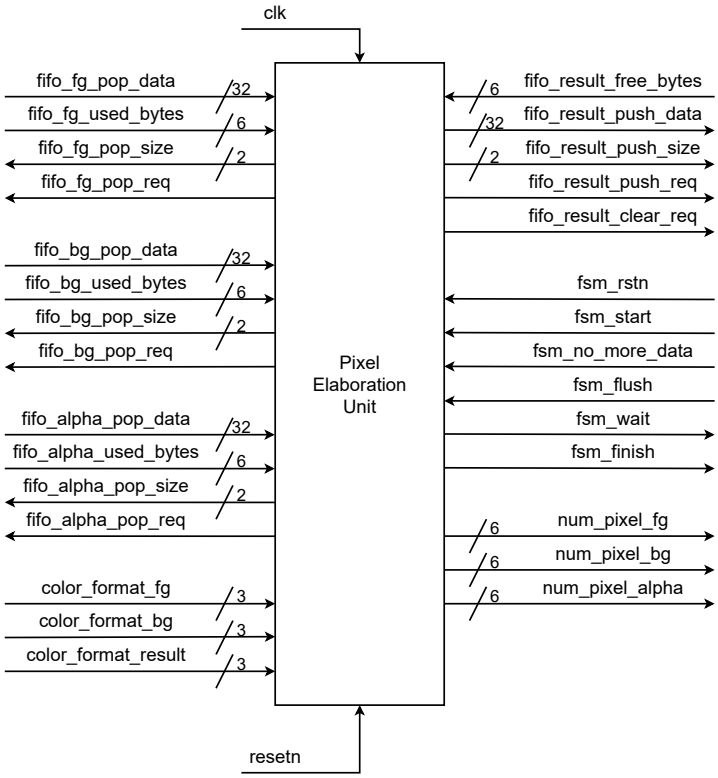

Figure 2.15: Pixel elaboration unit block

2.4.1 Color formats

In the specification document of the Engine, it was stated that the images elaborated by it could arrive with a color format falling into two main categories: 16-bit RGB565 and 32-bit RGBA. In the former, five bits are used to express the red component, six for the green one and another five for the blue one. In the latter, eight bits are used to express each color component. The letter A stands for the alpha channel where the range 0.0 to 1.0 is mapped to 0 to 255.

The memory representation of these formats can vary depending on the system, for example, in terms of endianness or the position of the alpha channel in a 32-bit format. In order to make the Engine compatible with the majority of the systems, support to six color formats has been implemented. Those formats can be seen in Figure 2.16. To easily distinguish each of them, a meaningful naming convention was adopted. Each color format is represented by a series of letters, three for 16-bit formats and four for 32-bit formats, followed by an equal amount of digits. The letters indicate the position of the colors in the data representation starting from the higher bits, while the digits indicate the number of bits used for each component.

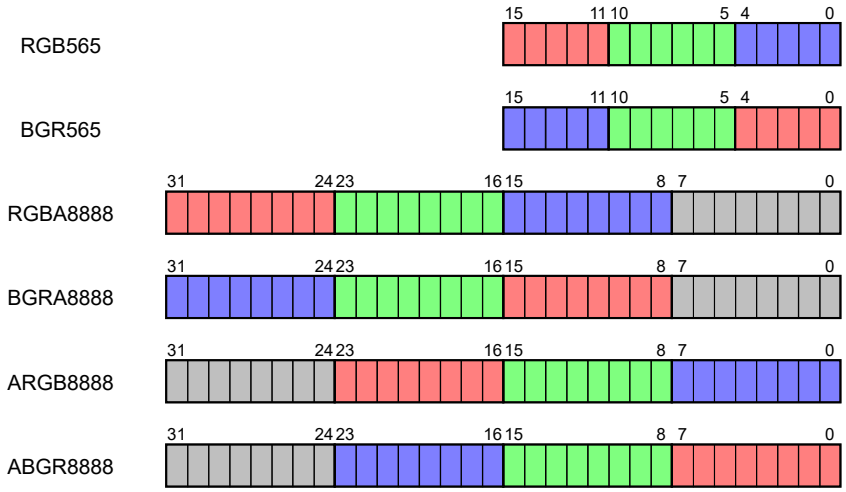

Figure 2.16: Color formats supported by the Stream Engine. The alpha channel is reported in gray.

Going from one format to another is quite simple when the two formats taken into account have the same bit width because it only requires reorganizing the bits in the pixel. When the two color formats have a different bit width, to have the same color expressed in both formats is crucial that for each color component, the most significant bits remain the same. This means adding some zeroes in the least significant bits when passing from 16-bit format to 32-bit or removing the least significant bits in doing the opposite. To understand it better, let us take the color "saddle brown" which has the following color components:

$$\text{Red} = 139 = 0\text{x}8\text{B}; \quad \text{Blue} = 69 = 0\text{x}45; \quad \text{Green} = 19 = 0\text{x}13$$

The alpha component can have an arbitrary value since it is not involved in the conversion. Let us assign it the value 255. Let us express this color first in RGBA8888, then, with a conversion, in RGB565 and then convert it back to RGBA8888. Those two conversions are graphically explained in Figure 2.17 where it is possible to notice both the bits removal and the addition of zeroes. It is clear that in going from a 32-bit format to a 16-bit one, there is a lack of precision in the color representation, even if the human eye cannot see the difference. Going in order from the top of the image, those three colors will be represented in memory as 0x8B4513FF, 0x8A22 and 0x884410FF.

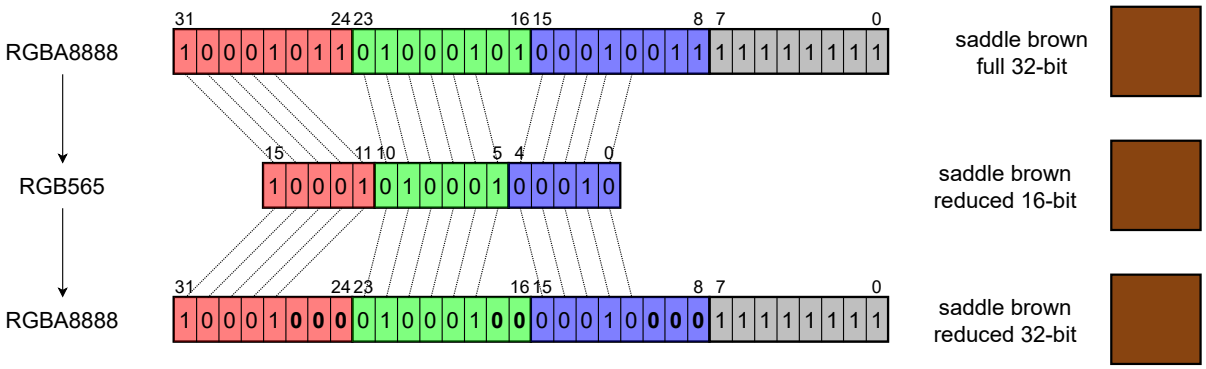

Figure 2.17: Color format conversion example

2.4.2 Datapath

The datapath of the pixel elaboration unit can be divided into two main parts. One is responsible for calculating the number of bytes to retrieve from input FIFOs and store in the result FIFO. The other one is responsible for pixel manipulation.

Let us now focus on the first one. A RTL representation is reported in Figure 2.18 and Figure 2.19. First, the widths in bytes of the input and output pixels are calculated from the color formats (6-way multiplexers). From the three obtained widths, the unit selects how many pixels it can elaborate simultaneously. The possible choices are two: if all the color formats are 2-bytes wide (RGB565 or BGR565), the unit can elaborate a number of pixels equal to the data width in bytes of the AXI4-Stream bus divided by two. Otherwise, it can elaborate only half of the previously calculated value. In the reported figures, the data width was set to 32 bits, so 4 bytes, the possible choices are 2 or 1 pixels at a time. From the obtained number of pixels per time, the maximum number of bytes to pull from the input FIFOs or to push to the result FIFO is calculated, taking into account the previously obtained widths of the pixels. For the FIFO containing the alpha parameters, the maximum number of bytes equals the number of pixels elaborated simultaneously. However, the actual number of pixels elaborated simultaneously can differ from the maximum calculated value at the end of the elaboration when the last pixels are sent to the Engine. Especially when the data width parameter assumes one of the higher values, the remaining number of pixels can be lower than the maximum, because of that the actual number of bytes to pull or push is selected between the maximum and the actual present in the FIFOs by the pixel elaboration unit FSM. All the selected number of bytes are then decreased by 1 to match the FIFO pop size and push size encoding.

The last task performed by this part of the datapath is to calculate the total number of pixels contained in each of the input FIFOs, including the number of alpha parameters since each one of them is used for a single pixel. Those calculated values are sent to the control unit block. Their usage will be discussed in the section regarding that block.

Let us now change the focus to the second part of the pixel elaboration unit, the one responsible for the pixel manipulation. Its RTL representation can be seen in Figure 2.20. It is composed of multiplexers that disassemble the data retrieved from the FIFOs. They separate the pixels according to the color formats and feed the single-pixel units. The

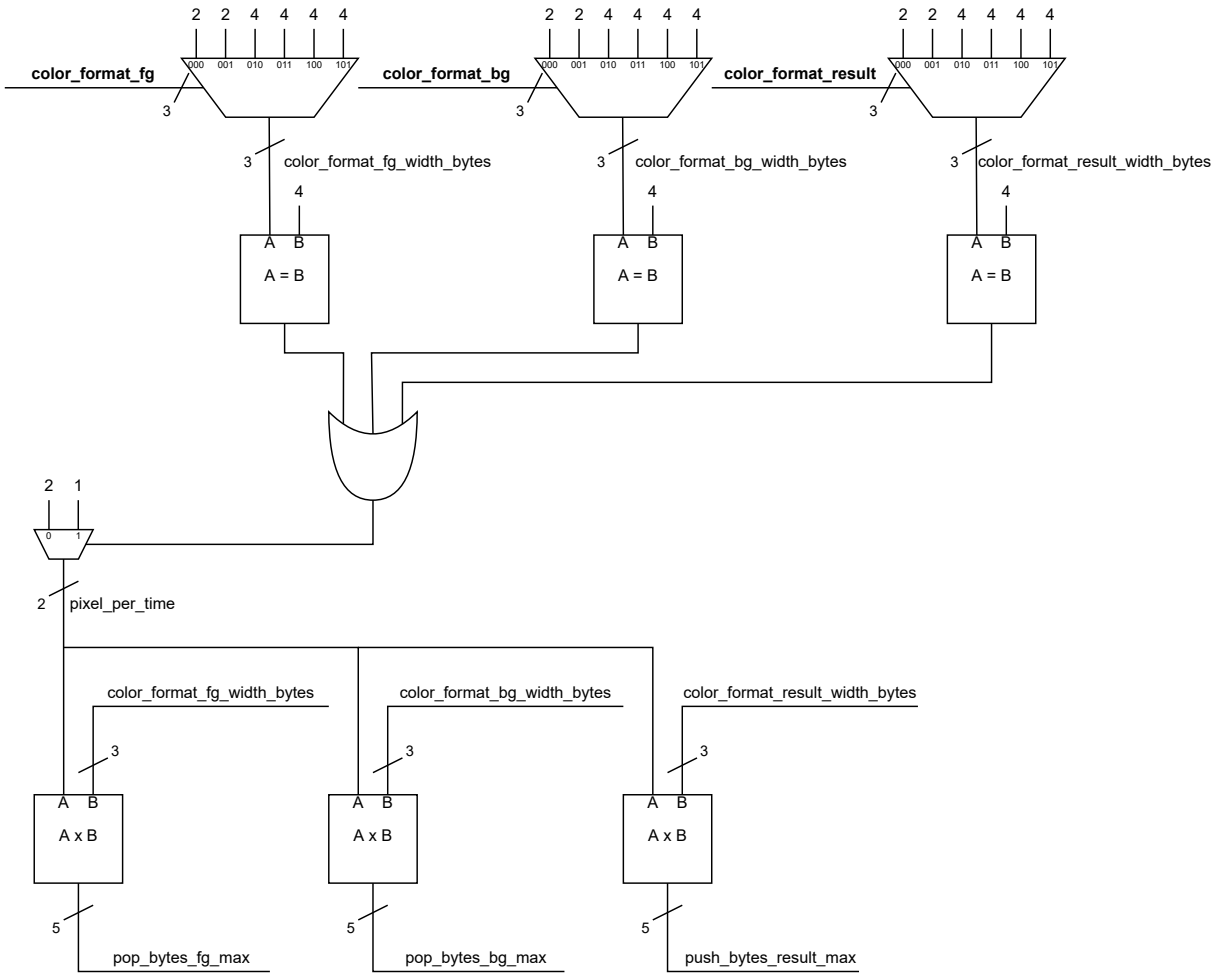

Figure 2.18: Pixel elaboration unit: datapath for calculating number of bytes to pull and push - part 1

actual elaboration is performed inside those units. Its functioning will be detailed later. The number of instances of those units equals the maximum number of pixels that can be elaborated simultaneously, even though not all of them are always used. If the actual number of elaborated pixels is half of the maximum value because of the color formats, only the first half of those units will be exploited. The other half will be fed with zeroes to avoid any data leakage. The outputs of the single-pixel units are reassembled by another multiplexer and sent to the result FIFO.

Single pixel unit

The single-pixel unit is the fundamental block to perform pixel manipulation. Its RTL internal structure can be seen in Figure 2.21 and Figure 2.22. First, it retrieves all the components from the two input pixels according to the respective color formats in a multiplexed way. Then it applies Equation 1.3 to the three color components in parallel. The Equation 1.3 has been tailored to be implemented using logic operators without

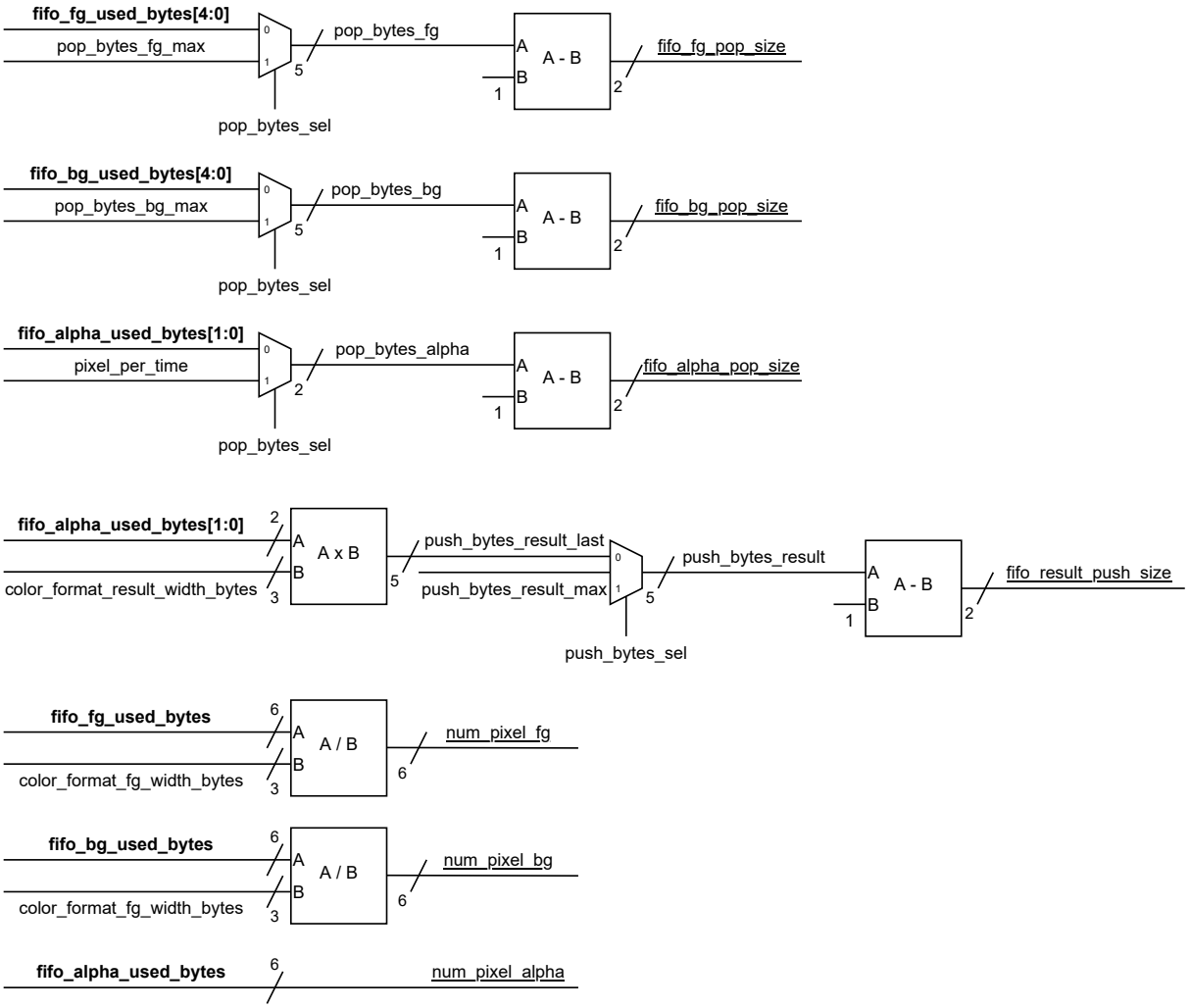

Figure 2.19: Pixel elaboration unit: datapath for calculating number of bytes to pull and push - part 2

wasting resources. It was reported before that for a memory representation, a pixel's alpha channel is given in a range from 0 to 255. The first change was then to multiply and divide Equation 1.3 by 255, applying the multiplication directly to alpha and then truncating the obtained alpha value:

$$C_O = \frac{C_A \lfloor 255\alpha_A \rfloor + C_B\big(255 - \lfloor 255\alpha_A \rfloor\big)}{255} \qquad \lfloor 255\alpha_A \rfloor = \alpha_A^I$$
$$= \frac{C_A \alpha_A^I + C_B\big(255 - \alpha_A^I\big)}{255}$$

Then it is possible to notice that $255 \simeq 256$ and dividing by 256 in hardware can be easily achieved by shifting the dividend by 8 bits to the right. So the final formula used in the single-pixel unit is:

$$C_O = \frac{C_A \alpha_A^I + C_B\big(255 - \alpha_A^I\big)}{256}$$

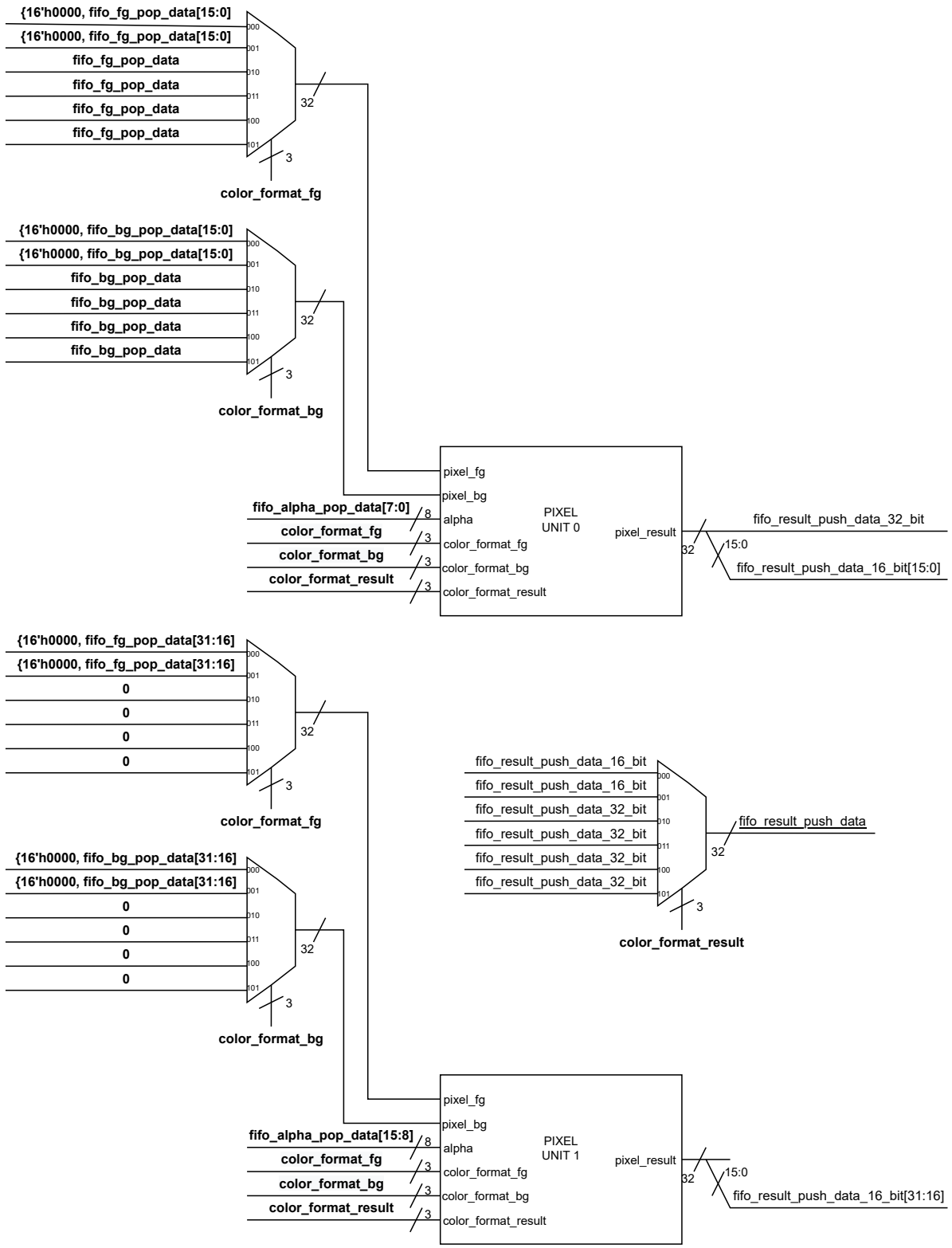

Figure 2.20: Pixel elaboration unit: datapath for performing the actual pixel elaboration

In Figure 2.21 can be noted the logic operators implementing the final formula. The subtraction $255 - \alpha_A^I$ has been replaced with a bit-wise NOT operator because this particular subtraction can be seen as the definition of the logic one's complement for an 8-bit unsigned number.

As requested in the design specification of the Engine, the internal processing is always made using 8 bits for each component. The computed color components are then reassembled by the multiplexer set reported in Figure 2.22 accordingly with the result color format. In case it is a 32-bit color format, the alpha component is set to 0xFF = 255 as requested by the specifications.

2.4.3 FSM

The main goal of the pixel elaboration unit FSM is to pull and push the correct number of bytes from all four FIFOs. Its state diagram can be seen in Figure 2.23. When the asynchronous reset is asserted, the FSM goes and remains in the `RESET` state. In this state the FSM clears the output FIFO by asserting `fifo_result_clear_req`. After the Stream Engine is released from asynchronous reset, the FSM moves to the `IDLE` state. Those two states are highlighted in green in the figure and represent the initial phase of the FSM.

During a complete alpha blending, the pixel elaboration unit FSM spends most of the time in the states `WAIT` and `PROCESS`. Those two states, highlighted in blue in the figure, represent the main phase of the FSM. The FSM first goes to the `WAIT` state from the `IDLE` state when it is started with the `fsm_start` signal coming from the control unit block. In this state, the FSM asserts `fsm_wait` to inform the control unit that the elaboration is paused because the input FIFOs do not contain enough data. When a number of pixels equal to the above described value "pixel per time", is present in all the input FIFOs and there is enough space in the result FIFO to store those pixels, the FSM moves to the `PROCESS` state where the pixel elaboration is performed through the assertion of `fifo_fg_pop_req`, `fifo_bg_pop_req`, `fifo_alpha_pop_req` and `fifo_result_push_req`. The number of bytes corresponding to the pixel per time is called "beat". It follows that two beats are double this amount. The FSM can stay in this state only if, at each iteration, there are two beats of data and space. This is due to the fact that the amount of data and space is calculated starting from the free bytes and used bytes signals of the FIFOs. It is important to remember that those signals do not consider an incoming pull or push request. It is recommended to set the FIFO depth Engine parameter to at least 2 to avoid the FSM going back and forth from processing to waiting.

When the control unit tells that the input FIFOs will not receive new data by asserting the `fsm_no_more_data` signal and the FSM is in the `WAIT` state, the FSM moves to a set of states to handle the end of the processing, `LAST_WAIT`, `LAST_PROCESS` and `FINISH`. Those states are highlighted in orange in the figure. The `LAST_PROCESS` state is entered if the three input FIFOs still contains some pixels and the output FIFO has enough space to store them. The `LAST_WAIT` state is entered if the output FIFO does not have enough space for the last pixels. The `FINISH` state is entered directly if there are no remaining pixels at the end of the elaboration. The `LAST_WAIT` state is similar to the `WAIT` state. When the output FIFO has enough data, the FSM moves from this state to the `LAST_PROCESS` state.

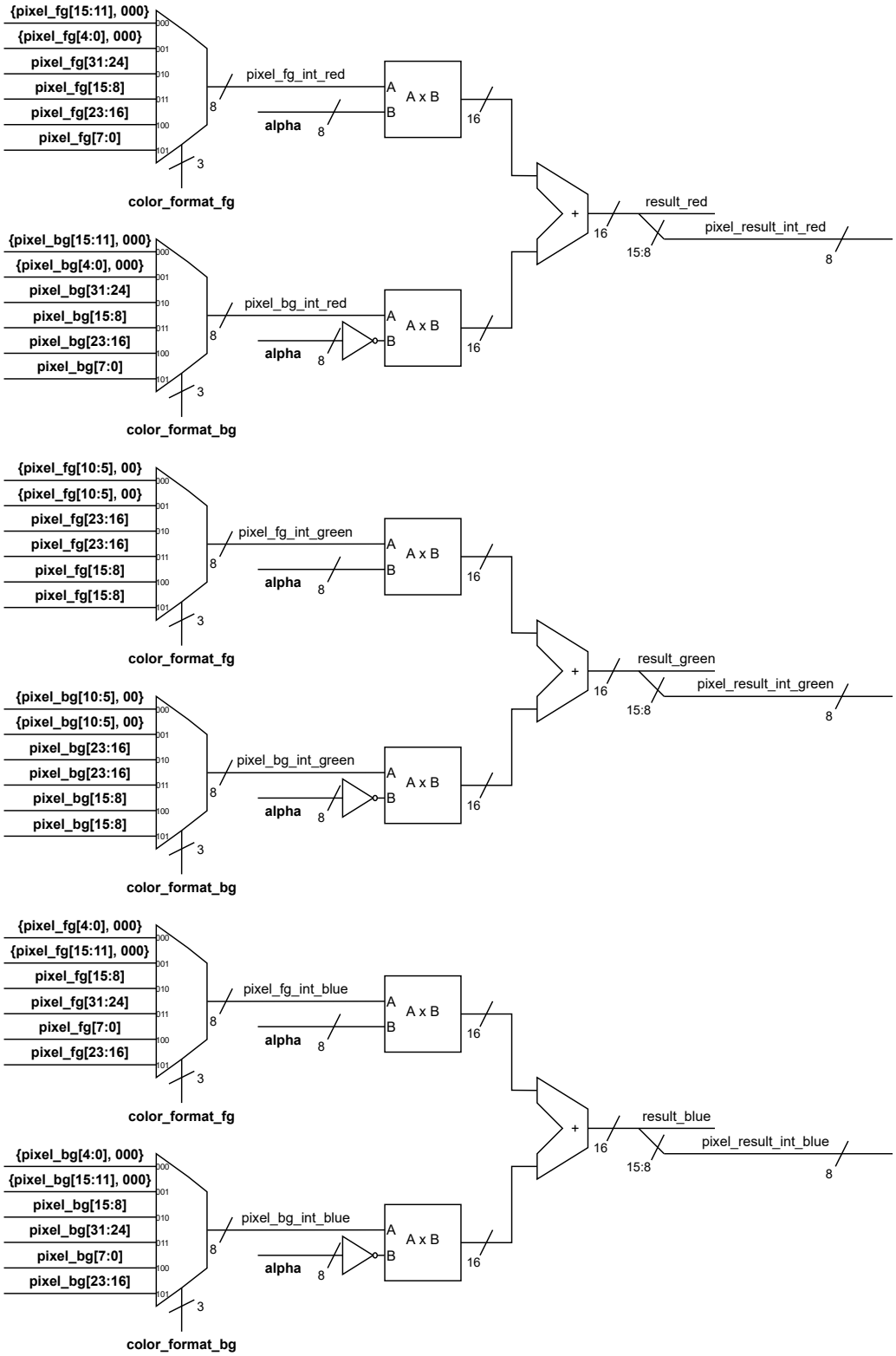

Figure 2.21: Single pixel unit, internal structure - part 1

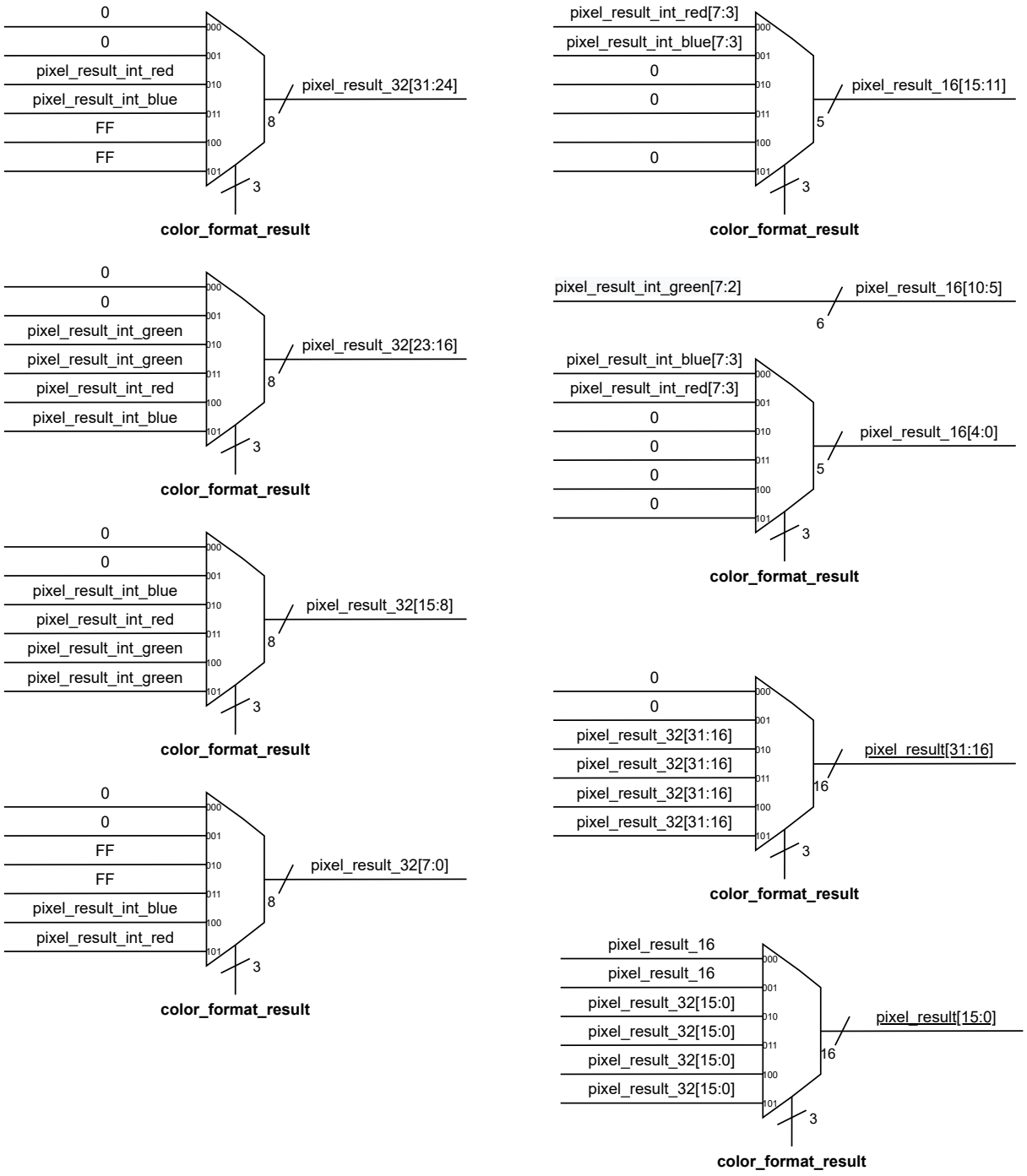

Figure 2.22: Single pixel unit, internal structure - part 2

The `LAST_PROCESS` state is similar to the `PROCESS` state, in addition to the signal asserted in that state, in this state `pop_bytes_sel` and `push_bytes_sel` are set to 0 to pop from the input FIFOs and push to the output FIFO the correct amount of bytes. From the `LAST_PROCESS` state the FSM moves directly to the `FINISH` state. In this state, the FSM asserts the `fsm_finish` signal to inform the control unit that the elaboration is completed.

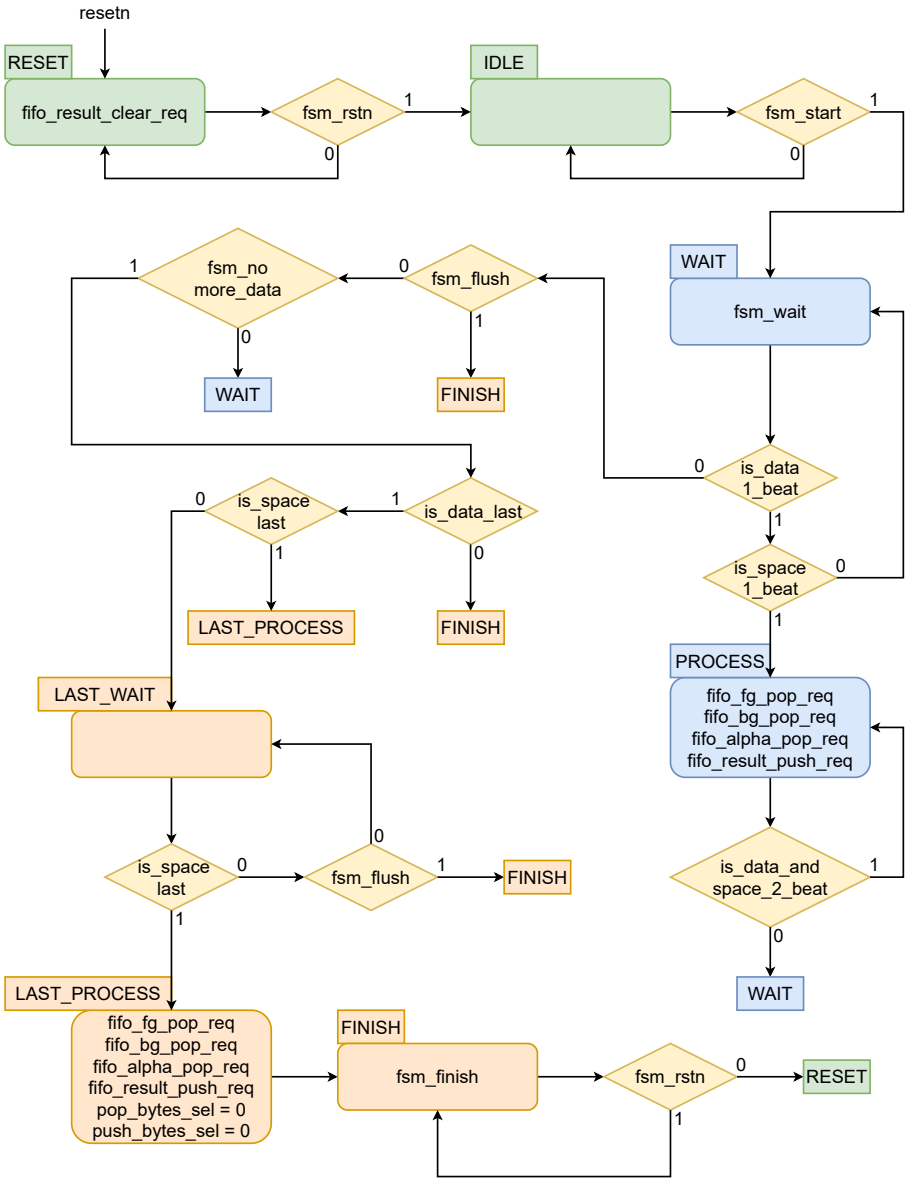

Figure 2.23: Pixel elaboration unit, FSM state diagram

The `fsm_flush` signal instead is used by the control unit when something goes wrong during the elaboration, and the ongoing process needs to be aborted. The FSM is sensitive to this signal when it is in the `WAIT` or `LAST_WAIT` states. In this case, it moves directly to the `FINISH` state.

All the condition signals used by the FSM are reported in Figure 2.24 and Figure 2.25. The condition `is_data_1_beat` is used to check if in the input FIFOs there is data for performing one complete elaboration while the condition `is_data_last` is used to check if there is still some data in the FIFOs to perform a last, reduced in number of bytes, elaboration. The condition `is_data_and_space_2_beat` is used to check if there is enough data in the input FIFOs and enough space in the output FIFO to perform two elaborations.

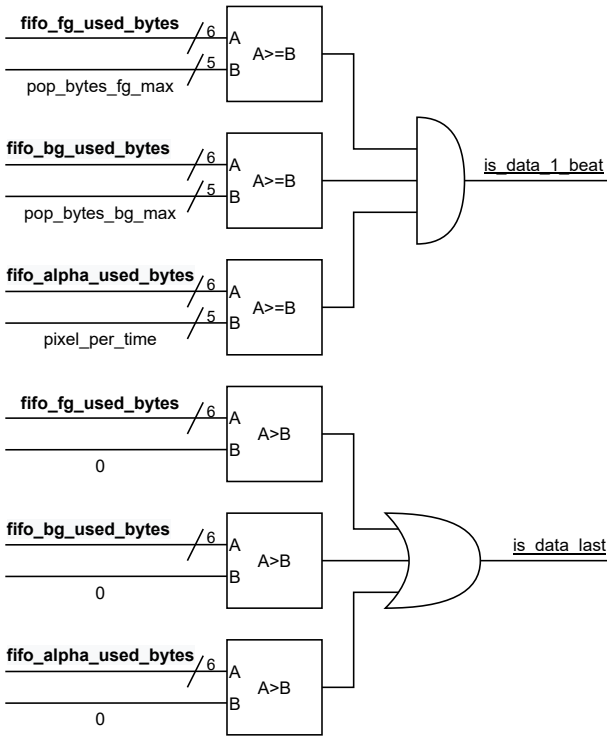

Figure 2.24: Pixel elaboration unit, FSM condition signal evaluation - part 1

The condition `is_space_1_beat` is used to check if there is enough space in the output FIFO to store a complete elaboration while `is_space_last` is used to check if there is enough space in the output FIFO to store the last elaboration.

2.5 Control unit

The control unit is the block that coordinates the operations of all the other blocks described. From the outside, it looks like as reported in Figure 2.26. It is composed of only an FSM with its condition signals evaluation logic.

Its design started by deriving all the phases required to perform an image composition from the specifications. When everything works correctly, the Engine should go through the following phases:

Reset All the FIFOs are cleared from the past image composition residual data in this phase.

Idle In this phase, the Engine input interfaces are ready to receive the three packets from the DMA.

Acquisition In this phase, the Engine receives the data. It also processes the data and sends them back to the DMA through the output interface.

Final processing Once the three packets are entirely received, the Engine continues to process and send data until the three input FIFOs are empty.

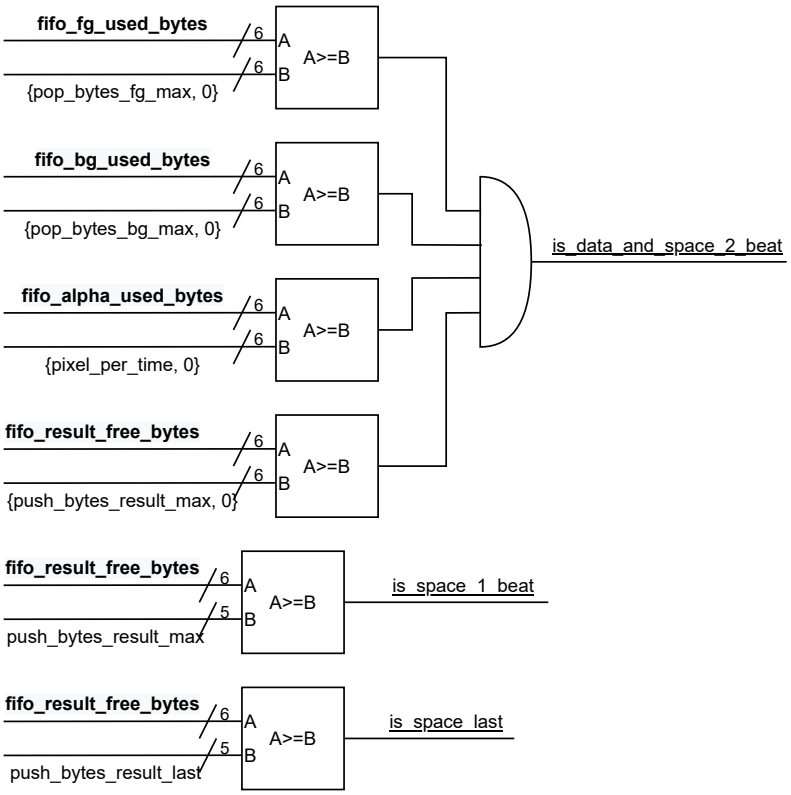

Figure 2.25: Pixel elaboration unit, FSM condition signal evaluation - part 2

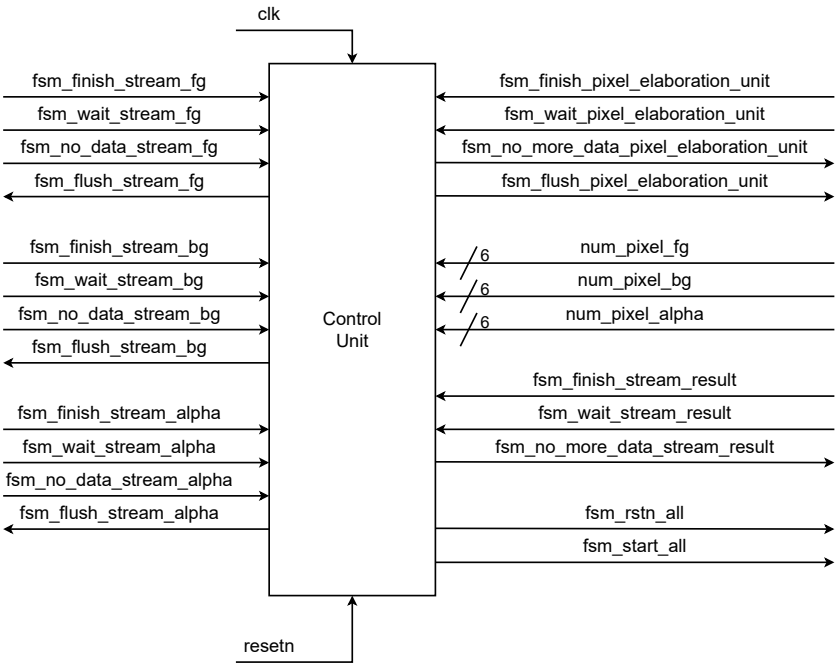

Figure 2.26: Control unit block

Final sending Once all the data is processed, the Engine sends data until the output FIFO is empty.

From those phases it was derived the FSM state diagram reported in Figure 2.27 together with the condition signals reported in Figure 2.28. Let us analyze it in detail. As long as the asynchronous reset of the Engine is asserted, this FSM remains in the `RESET` state. In this state, the FSM asserts the `fsm_rstn_all` signal that resets the FSMs of the other blocks. It then goes to the `IDLE` state where the other FSMs see the `fsm_rstn_all` deassertion and go to their idle state. The FSM then goes to the `START` state where it starts all the other FSMs by asserting the signal `fsm_start_all`. It remains in this state until all the stream receivers start receiving their packet. This is checked by the condition signal `is_all_data_started`.

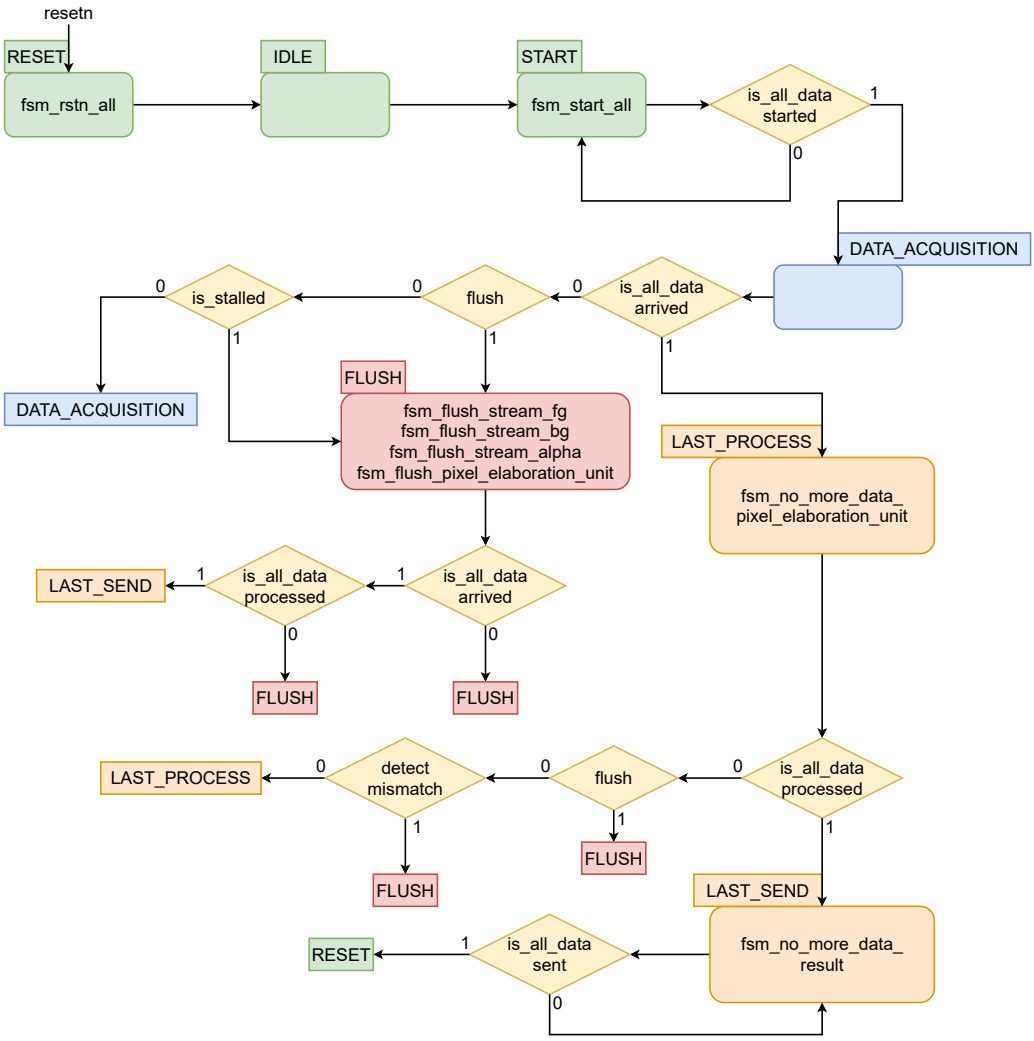

Figure 2.27: Control unit, FSM state diagram

After those initial states, the FSM goes and remains in the `DATA_ACQUISITION` state for most of the processing time. In this state, the stream receivers keep receiving their

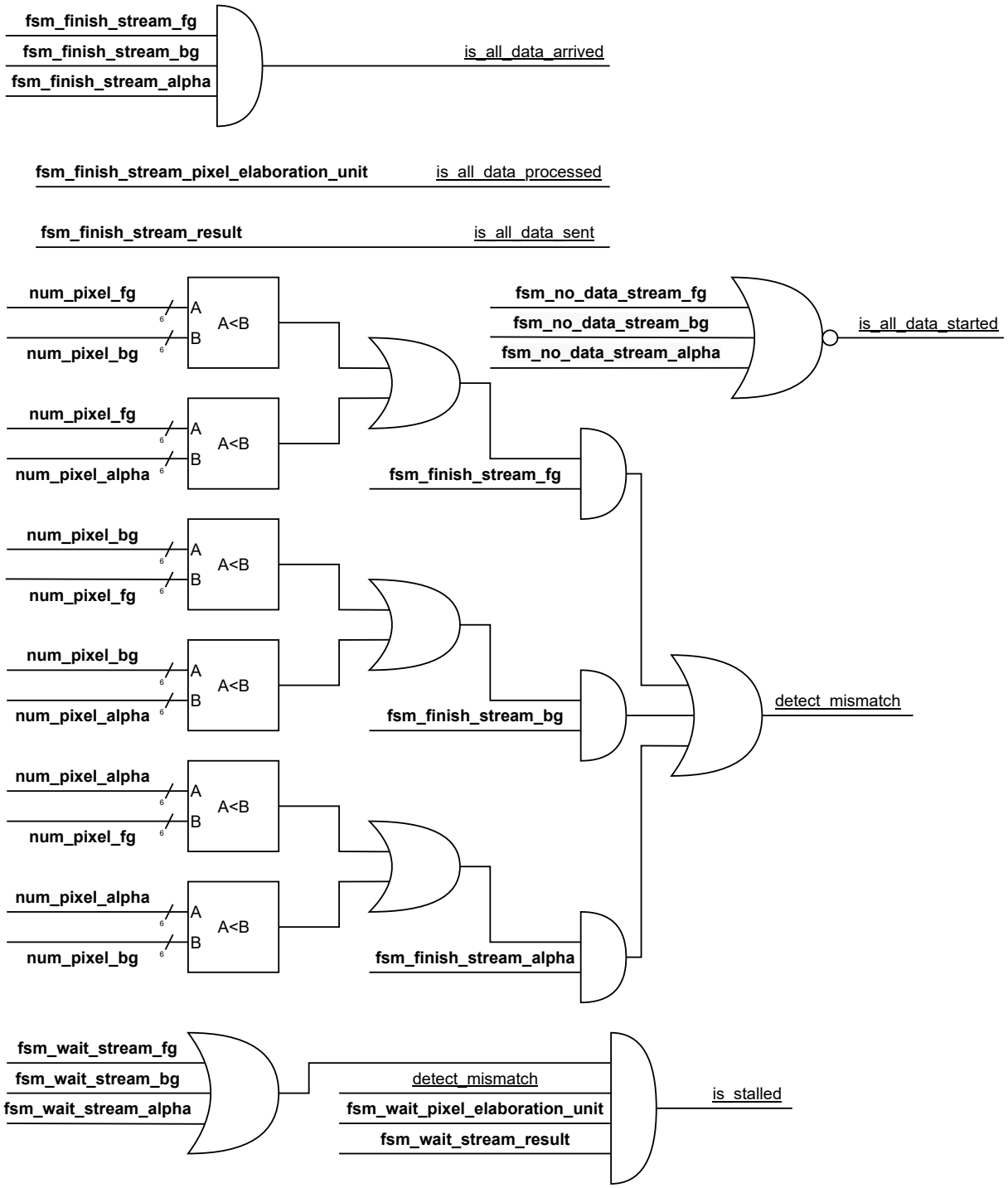

Figure 2.28: Control unit, FSM condition signal evaluation

packet, the pixel elaboration unit performs the pixel blendings, and the stream transmitter sends the result pixels back to the DMA. When all the stream receivers signal that they have finished the reception by asserting their finish signal (`fsm_finish_stream_fg`, `fsm_finish_stream_bg`, `fsm_finish_stream_alpha`), the FSM goes to the `LAST_PROCESS`

state where the pixel elaboration unit performs the last pixel blendings, and the stream transmitter keeps sending back the result pixels. In this state, the FSM asserts the `fsm_no_more_data_pixel_elaboration_unit` to tell it that those in the input FIFOs are the last pixels to process. When the pixel elaboration unit drains the input FIFOS and finishes all the blendings asserting its finish signal, the FSM goes to the `LAST_SEND` state where the stream transmitter keeps sending back the last pixels. In this state, the FSM asserts the `fsm_no_more_data_results` to tell the stream transmitter that those in the FIFO are the last result pixels to send. Once the stream transmitter finishes by closing the protocol with the DMA and asserts its finish signal, the FSM goes back to the `RESET` state, and a new image composition can start.

The `FLUSH` state was created to handle some faulty situations that can happen during the elaboration. The first one is related to the DMA itself. On the receiving side of the channel that accepts data from the Engine, there is a signal called `flush` connected to the Engine and in particular to this control unit. The channel can assert this signal as a hint to tell the Engine to send the last transfer of the packet as soon as possible because the data it is sending will be discarded. When the FSM is in the `DATA_ACQUISITION` state or the `LAST_PROCESS` state and detects the `flush` signal assertion, it goes to the `FLUSH` state where it tells the stream receivers and the pixel elaboration unit to flush everything. Once they have finished, the FSM goes to the `LAST_SEND` state to let the stream transmitter close the protocol.

Another situation that involves the `FLUSH` state is the mismatch condition checked by the `detect_mismatch` condition signal. Through the signals `num_pixel_fg`, `num_pixel_bg` and `num_pixel_alpha` the control unit is informed by the pixel elaboration unit about how many pixels are present in the three input FIFOs. A mismatch condition happens when one of the stream receivers has finished the reception of its packet and the pixels contained in its corresponding FIFO are less than those contained in one or both the other two input FIFOs. This mismatch happens because the channels have sent a different number of pixels from each other. When this FSM reaches the `LAST_PROCESS` state and detects a mismatch condition, it goes to the `FLUSH` state and continues from there as discussed for the `flush` signal.

The last situations that involves the `FLUSH` state is the stall condition checked by the `is_stalled` condition signal. It is similar to the mismatch. In fact, it also involves the `detect_mismatch` condition signal. However, it also needs the stream transmitter and the pixel elaboration unit to wait for data and the stream receiver to wait for space. To recover from this deadlock situation, the FSM can go from the `DATA_ACQUISITION` state directly to the `FLUSH` state and recover without any problem. In all the situations involving the `FLUSH` state, the correctness of the result cannot be guaranteed.

The only situation that can cause a denial of service in the Engine happens when at least one channel of the DMA has started sending its packet to the Engine, and at least one channel has not. In this case, the Engine will be blocked, waiting for all the input interfaces to terminate the protocol with the last transfer.

Chapter 3

Verification

In this chapter, it will be presented the unit-level verification phase of the Stream Engine using the *Universal Verification Methodology* (UVM). First, the UVM framework will be introduced. Then the realized test architecture will be described in detail. After that, the developed test cases will be presented, focusing on the aim of each test. Finally, the achieved results will be discussed to understand if the Engine can be considered sufficiently tested.

3.1 UVM

UVM is a standardized open-source methodology created for verifying integrated circuits (ICs). The idea is to verify the targeted device under test (DUT) by logical simulations where the DUT is fed with random constrained stimuli. The UVM framework eases the creation of those stimuli because it comes with classes and functions specialized in that. The outputs produced by the DUT are then compared against a reference model that replicates the DUT's functionalities.

Its source code is composed of classes written using SystemVerilog, which combines a hardware description language with an object-oriented one. With extended versions of some of them, those classes are connected to form a test architecture. Even though each class was created to perform a specific task inside a test, no strict rules exist to limit how those classes can be arranged. Because of that, users of this framework have created common patterns to organize those classes reasonably to obtain a tidy and clean architecture that can lead to code re-usability. Thanks to that, different tests to reproduce different scenarios can be run on the same architecture.

A UVM test is a logical simulation performed on a UVM compatible simulator. The simulation is divided into standard phases that are executed sequentially. Each of them performs a specific task required in the simulation. Those phases are reported in Figure 3.1. Let us quickly review them. In the `build` phase, all the test components are created. In the `connect` phase, the created components are connected. In the `end_of_elaboration` phase, it is possible to perform some refining to the test architecture before the actual simulation starts. The `start_of_simulation` phase is used to print the topology of the test architecture in addition to other relevant configuration information. The `run` phase

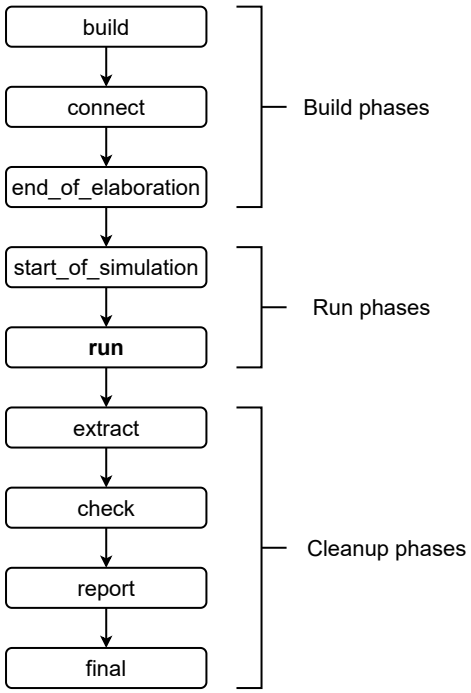

Figure 3.1: UVM phases

is the only one that consumes simulation time because, during it, the logical simulation is performed. In the `extract` phase, the results produced by the checking components are extracted. In the `check` phase, it is examined the DUT behavior and if the simulation ended correctly. In the `report` phase, the test results are displayed. If there is any action that remains to be done, it is performed in the `final` phase.

3.2 Test architecture

In this section, it will be described the test architecture built for testing the Stream Engine. Since it is formed by many nested blocks, the description will proceed incrementally in a bottom-up way.

3.2.1 Agents

In UVM, an Agent is a component responsible for driving one DUT interface and recording back its activity to share it with other components in the test. An example Agent is reported in Figure 3.2. It is composed of a Sequencer, a Driver and a Monitor. The Sequencer is the component that runs the sequences which contain the information on how to create the random constrained stimuli for the DUT. From those sequences, it extracts the transaction items containing the actual generated stimuli. The Driver receives the transaction items from the Sequencer and converts them into logical values that it can use to drive the DUT through an interface that is the point of connection between the test architecture and the DUT. The Monitor records the activity on the interface coming both from the Driver and

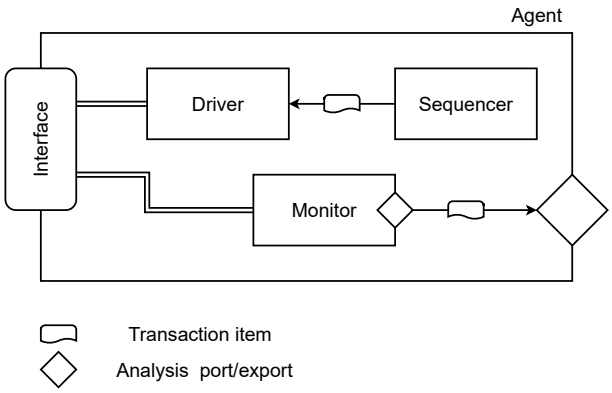

Figure 3.2: UVM generic Agent

the DUT. It then converts the activity in transaction items passed to other components through an analysis port, a UVM object that permits components to communicate.

Agents are usually protocol-specific because different protocols are based on different port lists. Thus, the agents' drivers and monitors need protocol-specific interfaces to interact with the DUT. In this test architecture, five types of Agents have been used: one for the clock interface, one for the reset interface, one for the AXI4-Stream interfaces, one for the GPI interfaces and one for the flush interface since the flush signal is not included in the AXI4-Stream protocol.

The AXI4-Stream Agents related to the foreground, background and alpha interfaces work actively. For them, the used sequences contain complete AXI-4 Stream packets like an entire image or the alpha data related to one. The Driver, in this case, is responsible for sending those packets to the DUT through the interface.

The AXI4-Stream Agent related to the result interface instead works reactively. It drives the related interface to accept the transfers coming from the DUT. From those transfers, it generates the transaction items containing the resulting images produced by the DUT. Those transaction items are then used to check the DUT behavior.

The Agent related to the clock interface uses a sequence containing the clock period and phase used to generate the clock signal for the DUT. The Agent related to the reset interface uses a sequence containing information on how to generate the DUT asynchronous reset like the duration and the type of deassertion, clock synchronous or asynchronous.

For the other Agents, the ones related to the GPI interfaces and flush interface, the used sequences contain only the signal value that the Drivers will send to the DUT as it is.

3.2.2 Virtual Sequencer

A Virtual Sequencer is a container component that holds a reference of each Sequencer in the test architecture. It is not a standard component, but it can be derived starting from the `uvm_sequencer` class. In Figure 3.3 is reported the one created for this architecture. The references will be linked to the actual Sequencers contained in the Agents. In this way, the sequencers are easier to be accessed when sequences have to be started upon them

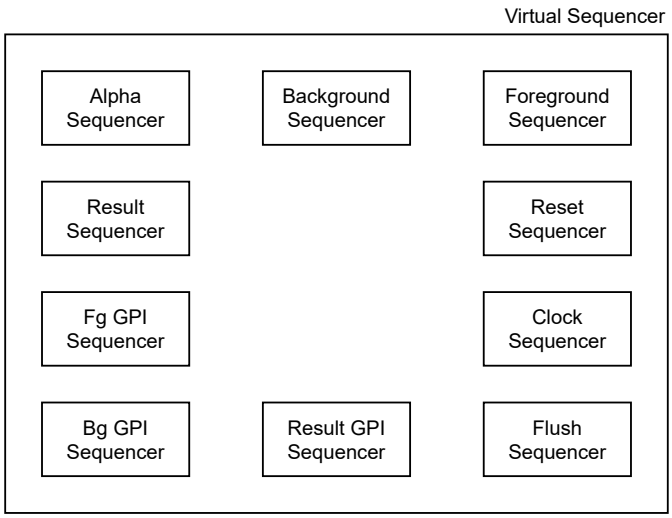

Figure 3.3: UVM Virtual Sequencer

because they are grouped in only one place.

3.2.3 Predictor

A Predictor is a component that contains the reference model against which the DUT is tested. It is not a standard component, but it can be created starting from the generic `uvm_component` class. In Figure 3.4 is reported the one created for this architecture. In the center of it, there is the reference model that, starting from each group of transaction items given to the DUT, produces a transaction item containing the predicted result. The produced transaction items are passed to the other components through the analysis port. The transaction items coming at the Predictor are stored in the analysis FIFOs. Those are UVM components with an unbounded size that can hold the transaction items indefinitely. In this way, the reference model can withdraw the transaction items when a complete set is available.

3.2.4 Scoreboard

A Scoreboard is a component that checks the DUT functionalities by comparing the transaction items coming from it against the ones coming from the Predictor. In Figure 3.5 is reported the one created for this architecture. Every time a result and a predicted result transaction item are available in the analysis FIFOs, it compares them. Depending on the obtained result, it increments a set of counters to compute a final report about how many errors it detected. Before making the comparisons, it also checks if the DUT received a flush command. In that case, it skips the comparison since the output coming from the DUT is probably faulty because of the flush. In the UVM `report` phase, it prints out its report containing how many comparisons were correct, how many were faulty and how many it skipped.

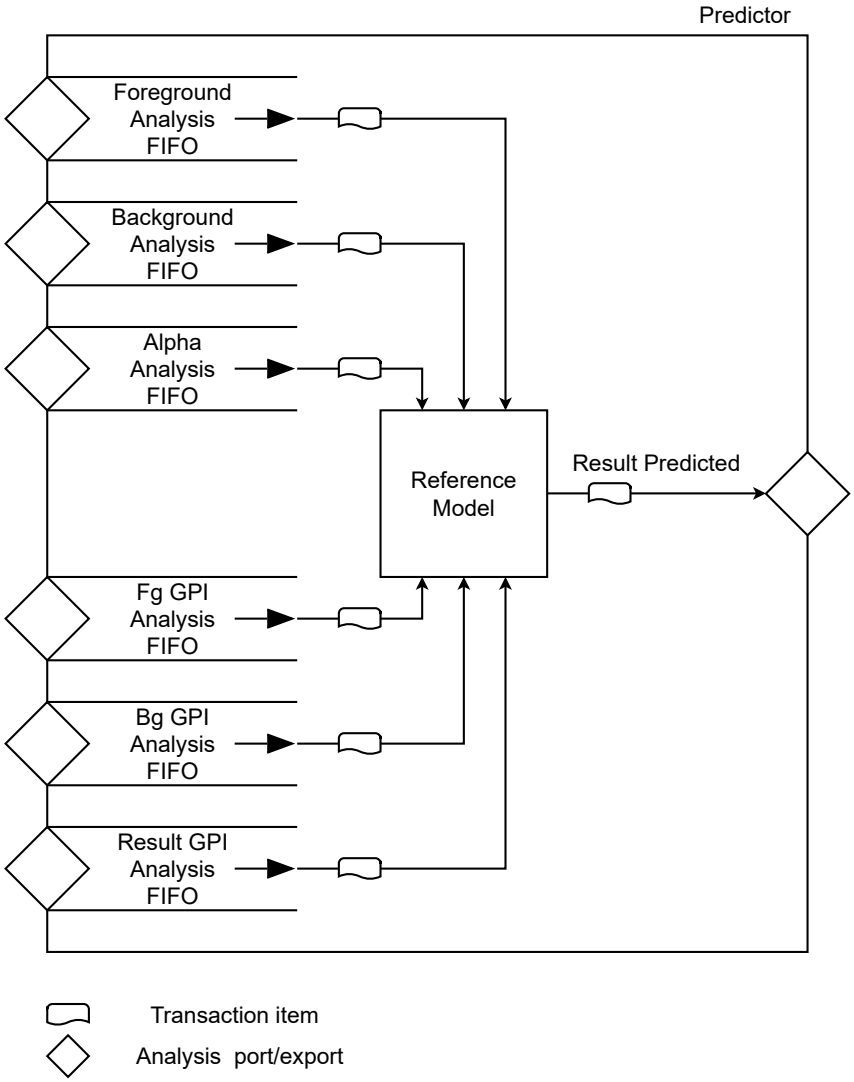

Figure 3.4: UVM Predictor

3.2.5 Coverage Collector

A Coverage Collector is a UVM component that samples information about DUT's functional coverage. Since it is not a standard component, it is derived from the `uvm_component` class. In Figure 3.6 is reported the one created for this test architecture. It is composed of four analysis FIFOs and a coverage model. The coverage model holds the information about the cover points that need to be recorded. For the Stream Engine, it was decided to collect information about the color formats. For each performed blending that is not affected by a flush, the occurred combination of the three color formats is stored. At the end of the simulation, it is possible to see if all the color format combinations have been tested sufficiently.

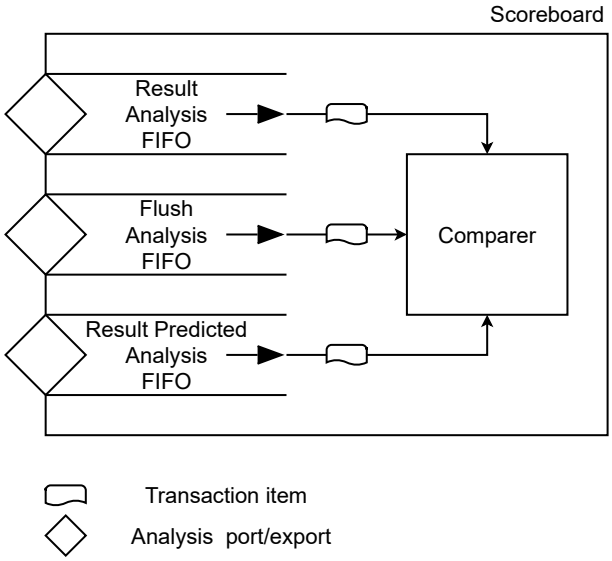

Figure 3.5: UVM Scoreboard

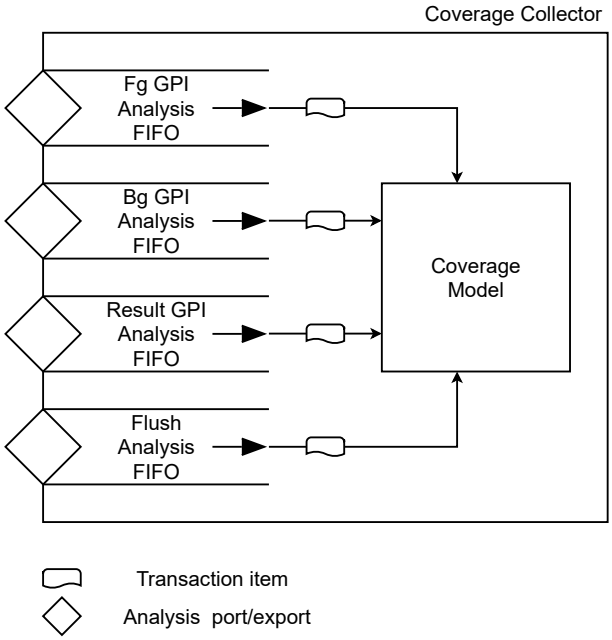

Figure 3.6: UVM Coverage Collector

3.2.6 Environment

The Environment component is a container that includes some of the already presented components. In Figure 3.7 is reported the one created for this test architecture. It was decided only to put the Predictor, the Scoreboard and the Coverage Collector connected in this Environment as illustrated. In this way, it will be easier to reuse this component in other UVM test architectures built to verify systems that include the Stream Engine. For the same reason, this Environment was given an analysis port to forward the transaction

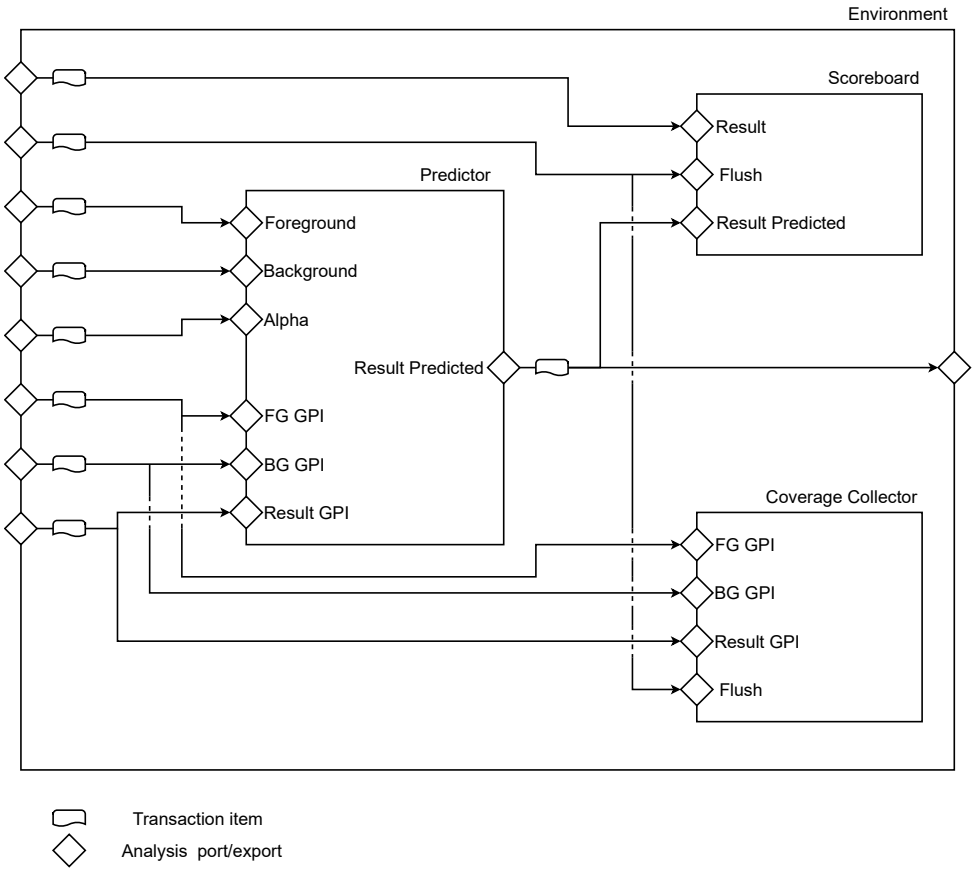

Figure 3.7: UVM Environment

items coming from the Predictor to another eventual DUT's Environment to build system-level Environments. This Environment is also configurable regarding the presence of the Scoreboard and the Coverage Collector. They can be independently disabled if not needed for the current simulation.

3.2.7 Integration Layer

The Integration Layer is the top component in the test architecture hierarchy. It is not a standard one, so it is created starting from the `uvm_component` class. In Figure 3.8 is reported the one created for this test architecture. It contains the Environment, the Virtual Sequencer and one Agent for each DUT interface. All those components are connected as illustrated. The analysis port of the Environment has not been represented because it is left unconnected for this architecture since there is no component to forward the transaction items containing the predicted results.

3.2.8 Testbench

After the presentation of all the UVM components and the UVM test architecture, it is now possible to look at the complete testbench for the unit-level verification of the Stream

Engine. A representation of it is reported in Figure 3.9. On the left side, there is the Stream Engine with all interfaces highlighted. On the right side, there is the `uvm_test_top` that is the actual test that will be performed. Apart from the Integration Layer, it contains all the sequences that will be used for the test. A description of its content will be given in the next section regarding the test cases. Each Stream Engine interface is connected to the corresponding Agent contained in the Integration Layer. On each of the four AXI4-Stream interfaces is connected a protocol checker (PC). It is not a UVM component, and because of that, it is put outside the `uvm_test_top`. Its task is to supervise the AXI4-Stream interface and check if all the communications are protocol compliant.

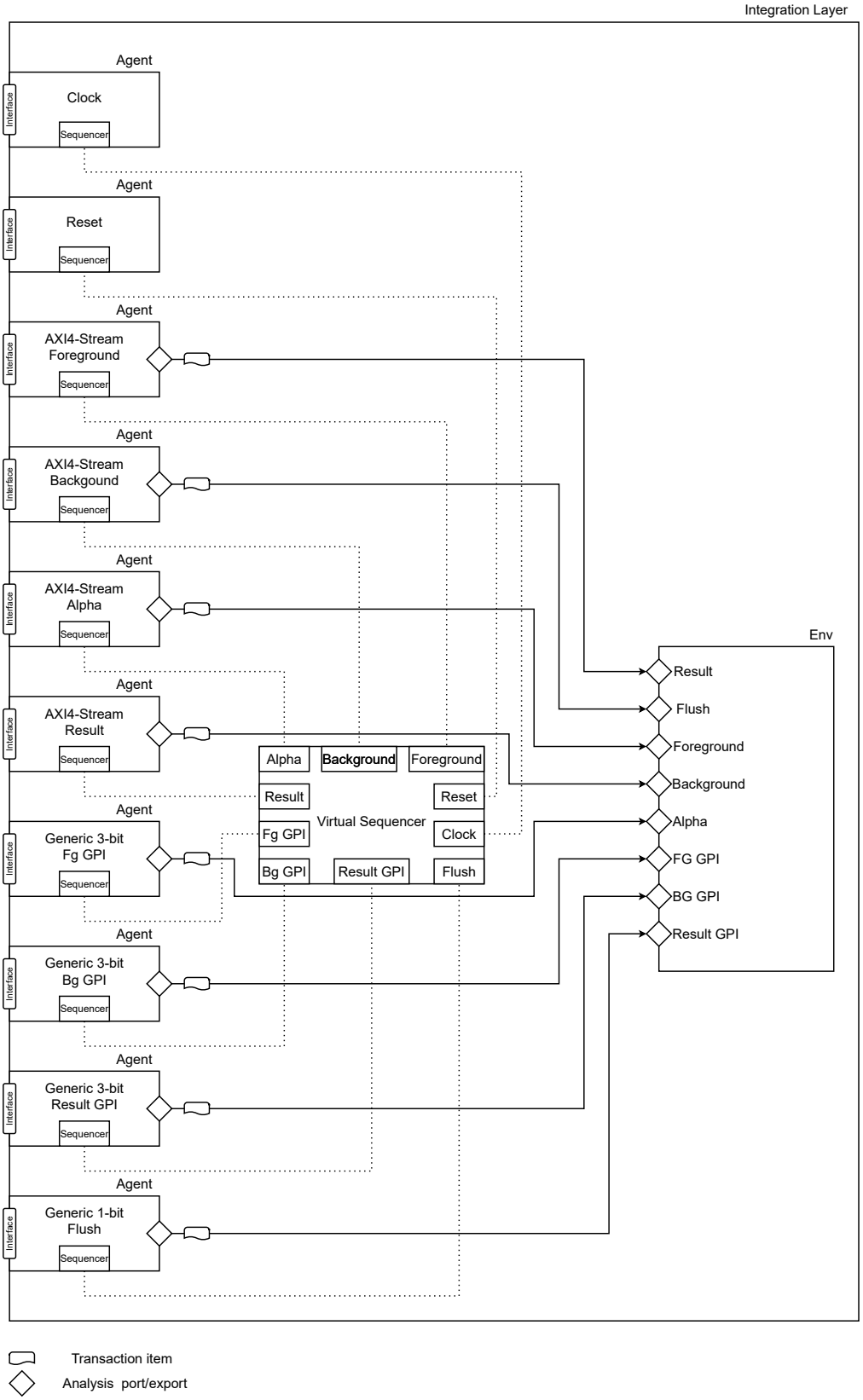

Figure 3.8: UVM Integration layer

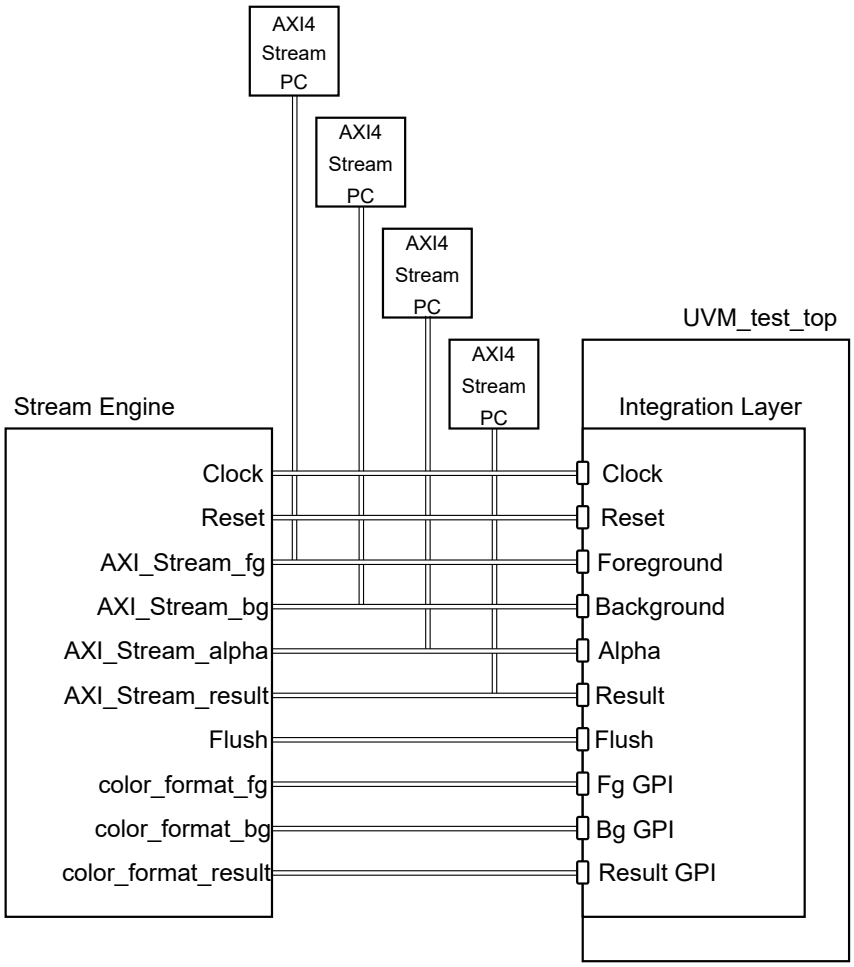

Figure 3.9: UVM complete Testbench

3.3 Test cases

This section will describe the UVM test cases developed for the Stream Engine Verification. Their creation progressed incrementally, starting from simple scenarios to the most complex ones.

3.3.1 Base test

The base test is not an actual test. Since SystemVerilog is an object-oriented language, it was chosen to create a virtual base test class and derive the actual tests from it. It contains the attributes and methods common to all the tests.

During the UVM build phases, this class is responsible for the test initialization. First, it configures all the Agents and the UVM Environment. There is a set of configuration parameters for the AXI4-Stream Agents and the UVM Environment that can be changed by the actual tests. They are reported in Table 3.1. After that, it sets a reactive sequence

for the AXI4-Stream result Sequencer. In this way, the AXI4-Stream result Agent can act as an active receiver. Finally, it performs the DUT reset by sending a reset sequence to the reset Sequencer.

Table 3.1: Configuration parameters common to all tests.

Parameters	**Description**
`tvalid_fg_delay_min`	For each of the three active AXI4-Stream Agents, there is a couple of parameters related to the `tvalid` signals behavior, a minimum and a maximum. For each transfer, the Drivers in those Agents pick a random number between the minimum and the maximum. It will represent the number of cycles the Driver has to wait before sending a new transfer.
`tvalid_fg_delay_max`	
`tvalid_bg_delay_min`	
`tvalid_bg_delay_max`	
`tvalid_alpha_delay_min`	
`tvalid_alpha_delay_max`	
`tready_result_delay_min`	A similar thing is done by the reactive result Agent with the `tready` signal. In this case, the random number represents the number of cycles the Driver has to wait before asserting the signal again after having accepted the previous transfer.
`tready_result_delay_max`	
`has_scoreboard`	With this flag is possible to enable the presence of the Scoreboard in the UVM Environment.
`has_coverage`	With this flag is possible to enable the presence of the Coverage Collector in the UVM Environment.

This class also contains two methods for performing all kinds of operations on the DUT. One method is called `generate_1_blend`. It accepts the number of pixels for the blending as an argument. First, it generates a random combination of three color formats, one for the foreground, one for the background and one for the result. Then it sends the color formats to the DUT through the GPI Sequencers. From the generated color formats and the number of pixels argument, it calculates the number of bytes to set in each AXI4-Stream sequence. Those sequences will then be started upon their respective Sequencers. In this way, a correct blending can be sent to the DUT. The method can also accept two more number of pixels arguments. There will be three numbers, one for the foreground image, one for the background image and one for the alpha. In this way, it is possible to send a mismatched blending to the DUT by setting three different values for the numbers. The other method is called `generate_1_flush`. It interacts with the current blending by sending two flush sequences, one for asserting the flush signal when the Stream Engine starts sending out the result packet, the other for deasserting the flush signal after the Engine has sent the last transfer of the packet.

3.3.2 Hello world

The hello world test was the first actual created test. It recalls the base test operations. That means that the only generated sequence is the reset one. It was created to have a quick first example to check if the test architecture and connections were correctly put together. Its results are inspected directly by looking at the simulator waveforms. This test is passed if the Stream Engine enters and exits the reset phase correctly with the AXI4-Stream interfaces waiting for a new blending to start.

3.3.3 Single blend

The single blend is a test designed to perform one elaboration on the Stream Engine. It comes with other two additional parameters with respect to the base test, `num_pixel` and `flush_transaction`. The first sets the number of pixels of the blending. The second is a flag to decide if a flush should be sent to the Stream Engine for that blending. The results of this test are checked by reading the Scoreboard summary at the end of the simulation. It should show one passed or skipped comparison according to the `flush_transaction` flag. This test aims to check that the Engine behaves correctly when performing a blending.

3.3.4 Multiple blends

The multiple blend is a test designed to perform a series of elaborations on the Stream Engine. It comes with four additional parameters with respect to the base test, `num_blend`, `min_pixel`, `max_pixel` and `percentage_flush`. With `num_blend` is possible to configure how many elaborations the test has to perform on the Stream Engine. For each elaboration the number of pixels for the blending is chosen randomly in the range defined by `min_pixel` and `max_pixel`. The last parameter, `percentage_flush`, determines the percentage of blendings that will be affected by a flush. The results of this test are checked by reading the Scoreboard summary at the end of the simulation. It should show a total number of

comparisons equal to num_blend roughly divided between passed and skipped according to percentage_flush. This test aims to check that the Engine behaves correctly when a series of blendings is performed. In particular, it is important to check that the Engine does not get stuck when flushed blendings are mixed with non-flushed blendings.

3.3.5 Single blend with mismatch

The single blend with mismatch is a test designed to perform one faulty elaboration on the Stream Engine. It comes with with four additional parameters with respect to the base test, num_pixel_fg, num_pixel_bg, num_pixel_alpha and flush_transaction. With the first three parameters is possible to configure the number of pixels for each AXI4-Stream input individually. The last one is a flag to decide if a flush should be sent to the Stream Engine together with the faulty blending. The results of this test are checked by reading the Scoreboard summary at the end of the simulation. It should show one failed or skipped comparison according to the flush_transaction flag. This test aims to check that the Engine can detect a faulty elaboration and recover from it even when mixed with a flushed elaboration.

3.3.6 Multiple blends with mismatch

The multiple blends with mismatch is a test designed to perform a series of faulty elaborations on the Stream Engine. It comes with four additional parameters with respect to the base test, num_blend, min_pixel, max_pixel and percentage_flush. Those four parameters can be used the same way as in the multiple blends test. In this case, num_blend indicates the number of faulty elaborations performed on the Stream Engine. The results of this test are checked by reading the Scoreboard summary at the end of the simulation. It should show a total number of comparisons equal to num_blend roughly divided between failed and skipped according to percentage_flush. This test aims to check that the Engine behaves correctly when a series of faulty blendings is performed. In particular, it is important to check that the Engine does not get stuck and recover correctly from all the faulty and flushed blendings executed.

3.3.7 Multiple blends all types

The multiple blends with all types is a test designed to perform a series of elaborations on the Stream Engine. Those elaborations can be correct, faulty or flushed. It comes with five additional parameters with respect to the base test, num_blend, min_pixel, max_pixel, percentage_flush and percentage_mismatch. The first four parameters can be used the same way as in the other multiple blends tests. The last one, percentage_mismatch determines the percentage of blendings that will be created faulty. To keep track of the number of the different generated blendings, the test contain three counters, expected_num_pass, expected_num_failed and expected_num_skip. A the end of the simulation, those counters are checked against the Scoreboard report. The test is passed if the comparison division reported by the Scoreboard matches the one reported by the counters. This test

aims to check that the Engine behaves correctly in all possible scenarios. In particular, it is important to check that the Engine never gets stuck in one of the blendings.

3.4 Results

Running the tests described above led to some modifications of the Stream Engine. Each time an improved version of the Engine was obtained, all the tests were performed again to find other flaws. During those tests, many possible values for the parameters `data_width` and `fifo_depth` were tried to test all the possible configurations of the Engine. In the chapter regarding the design, it was already reported the last Stream Engine version. Once it was obtained, a final series of tests were performed to confirm the correct functioning of this version in all possible scenarios. So, the last presented test was taken, the multiple blends with all types, and was performed seven times, each time with a different set of simulation parameters. The values of the parameters at each iteration are reported in Table 3.2. The first iteration was the default one. In the second, the AXI4-Stream Agent related to the foreground was set to send its packet slower than the others. In the third, the slower one was the background Agent. In the fourth one, it was the alpha Agent. In the fifth one, the AXI4-Stream Agent related to the result was slower in receiving the packet. Those last four iterations recreated scenarios where one channel, for any reason, cannot send or receive data at the same pace as the others. The sixth and seventh iterations were needed to reach some corner cases that required a low number of pixels and, in turn, no flush or mismatch blendings.

Table 3.2: Configuration parameters for the coverage tests. The underlined values are the ones that differ from the default represented by the first iteration.

	Values at each test iteration						
Parameters	**1**	**2**	**3**	**4**	**5**	**6**	**7**
`num_blend`	1000	1000	1000	1000	1000	1000	1000
`min_pixel`	100	100	100	100	100	1	1
`max_pixel`	150	150	150	150	150	20	20
`percentage_flush`	20 %	20 %	20 %	20 %	20 %	0 %	20 %
`percentage_mismatch`	20 %	20 %	20 %	20 %	20 %	20 %	0 %
`tvalid_fg_delay_min`	0	0	0	0	0	0	0
`tvalid_fg_delay_max`	4	20	4	4	4	4	4
`tvalid_bg_delay_min`	0	0	0	0	0	0	0
`tvalid_bg_delay_max`	4	4	20	4	4	4	4
`tvalid_alpha_delay_min`	0	0	0	0	0	0	0
`tvalid_alpha_delay_max`	4	4	4	40	4	4	4
`tready_result_delay_min`	0	0	0	0	0	0	0
`tready_result_delay_max`	4	4	4	4	20	4	4
`has_scoreboard`	True	True	True	True	True	True	True
`has_coverage`	True	True	True	True	True	True	True

The Scoreboard report was checked to confirm the test's success each time it was performed. For each iteration, it was also recorded the functional coverage performed by the Coverage Collector and the code coverage performed by the simulator. The coverage data files obtained from each simulation were merged to perform a comprehensive coverage analysis. The functional coverage showed that the simulations several times hit all the color format combinations. For the code coverage analysis, it was created a bar chart reported in Figure 3.10. It contains coverage information regarding different features about the four blocks developed for the Stream Engine. The *Branches* feature measures the coverage of the directions taken when a branch is encountered in the code. The *Expression* feature measures the contribution of each part of a logical expression in determining the value of 1-bit signals such as condition signals for the FSMs. The *FSM States* feature measures how many states were reached for each FSM. The *FSM Transitions* feature measures the coverage of the transitions among the possible ones. The *Statements* feature measures how many lines of code were reached during the simulations.

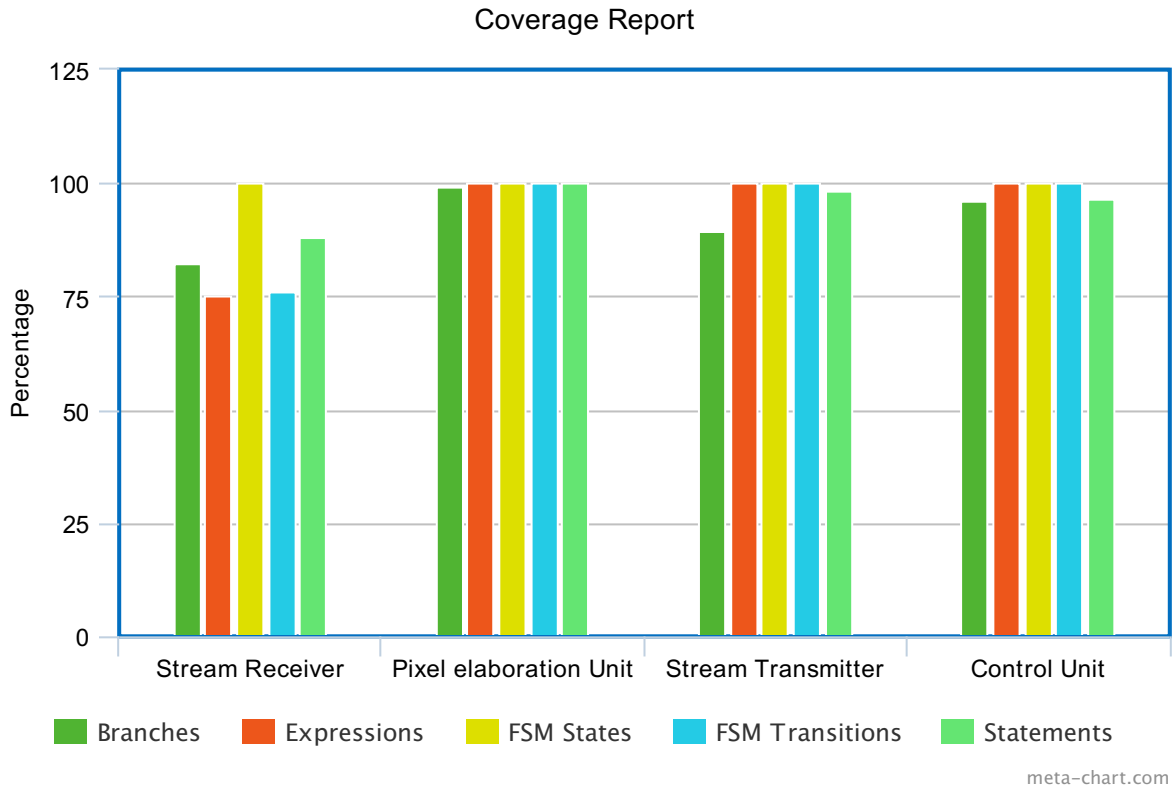

Figure 3.10: Code-coverage report for the developed blocks

Among the four blocks analyzed, the less covered seems to be the Stream Receiver, with some of its features reaching just 75 %. Those coverage holes were analyzed in detail, and they are mainly caused by a few FSM transitions that are difficult to reach. They cannot be removed from the FSM because they are needed for compatibility with the DMA specification. To reach those FSM transitions, it is necessary to create specific unconventional sequences for the active AXI4-Stream agents.

After all those tests, it was possible to declare the Stream Engine enough tested and ready to be integrated next to the DMA controller.

Chapter 4

System integration

This chapter will present the integration test of the designed Stream Engine for alpha blending using an environment that emulates a system-level scenario for the DMA Controller. In the first section, it will be described the environment architecture and how it works. The developed test cases will be presented alongside the obtained results in the second section.

4.1 Testbench Architecture

The project where the DMA controller was developed also contains an environment that lets users try out the DMA features using high-level tests written in C language. Since this environment was not developed as part of the Stream Engine work, not all parts will be described. Only the ones needed for testing the Engine will be analyzed.

In Figure 4.1 is reported the architecture of this environment with a focus on the parts needed to test the Engine integration. In the center of it, there are the DMA and the Stream Engine. They are connected in the same way as in the first chapter, where it was illustrated an example connection. The DMA is also connected to two more components, a memory and a Command Generator. The first emulates the main memory of a computing architecture. In a typical system, the DMA moves data to or from it according to the received instructions. The second is the component responsible for generating the commands for the DMA. It emulates the elaboration unit that sends the commands to the DMA. All those components are instantiated by a testbench that, in addition, generates the clock and the reset signals.

The simulation is carried out using a logical simulator, the same used for the UVM verification. However, a few steps are required to translate the developed test into DMA commands before launching it. First, the test is written in C language using a specific library created for the DMA. The obtained C file should contain an ordered sequence of commands to be executed from the DMA. This file is then compiled and executed. The execution output is a text file containing the DMA registers' values that should be written to perform the commands. Peripherals like the DMA need their internal registers to be configured to make them work. The text file is then passed to the Command Generator that will open and use it during the logical simulation to perform the writes on the DMA

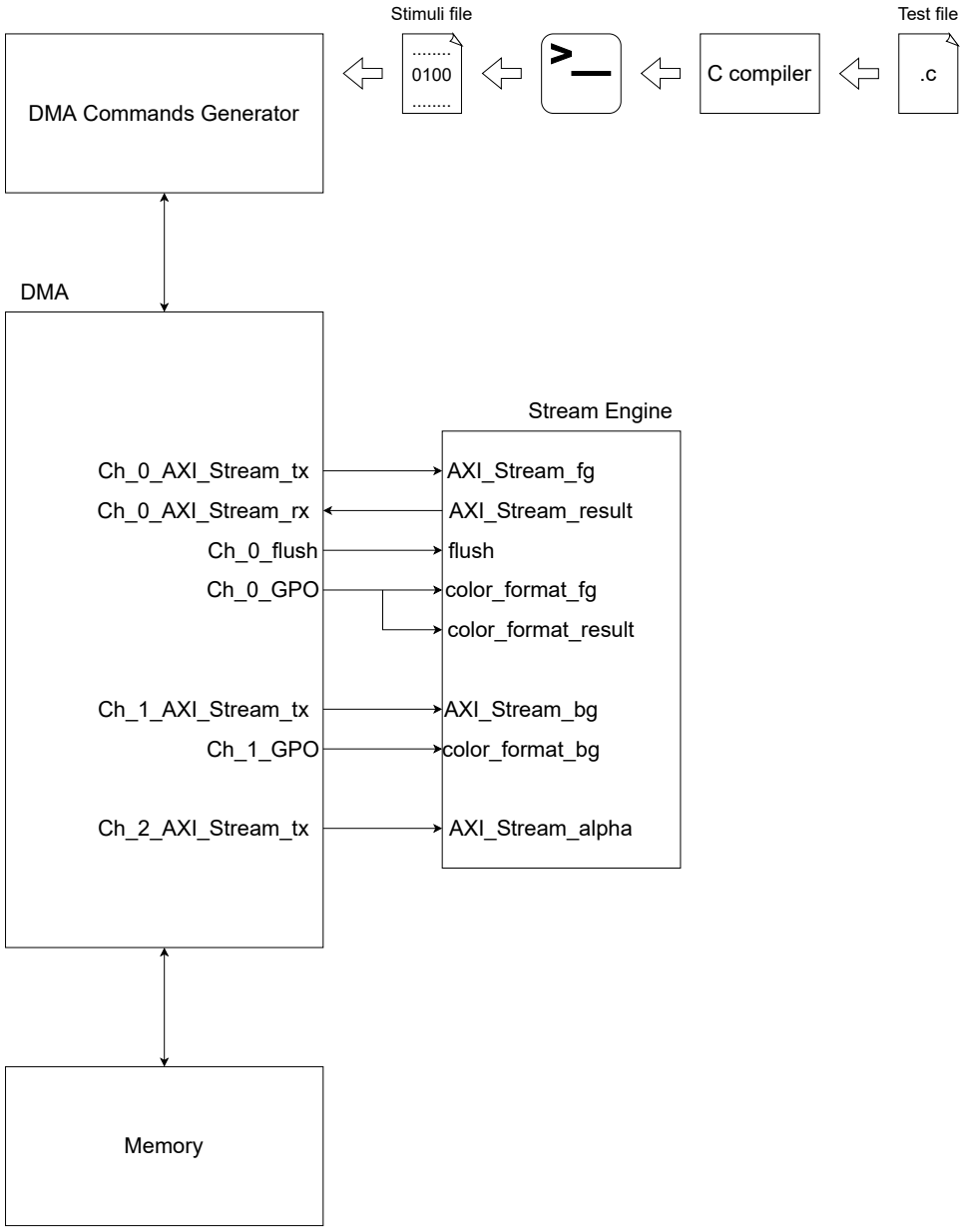

Figure 4.1: System-level testbench architecture

registers.

4.2 Test Cases

Using the presented environment, two tests were created to check the Stream Engine's correct functioning when it receives data from the DMA. Those tests aim to replicate real scenarios where the DMA is used not only to move data but also for performing alpha blendings using the streaming feature. Although the environment contains an automatic result checking, this does not cover the alpha blending operations. For each of the two

tests, a different method was adopted for confirming their success.

4.2.1 Stability test

The first developed test was named *Stability test* because it aims to check that the DMA can execute correctly a series of commands involving the Stream Engine. It also verifies that the DMA, the Engine and all the other components in the testbench are correctly connected.

The C file containing this test is organized in the following way: on the top part, it has three configuration parameters, one for choosing how many alpha blendings have to be performed, two for selecting the range of pixels for each blending.

After the configuration part, there is the part of the commands. First, the test randomly selects the number of pixels for the current blending using the given range. It also randomly selects three different color formats among the six available, one for the foreground, one for the background and one for the result.

After that, the test sets channel 0 of the DMA to perform the first command. It programs the source registers of this channel with the starting address and the length in bytes of the foreground image. The destination registers are programmed with the same information regarding the resulting image. In this way, the channel can pick the first image and store the result. The test also enables the streaming interface of the channel, and the GPO register is filled with the color formats corresponding to the two images. After the configuration, channel 0 is started.

The test moves to set channel 1 of the DMA to perform the second command. The test programs the source registers of this channel with the starting address and the length in bytes of the background image. The destination register regarding the length is set to 0. In this way, the channel will only pick the background image without storing anything back. The test also enables the streaming interface of the channel, and the GPO register is filled with the color format corresponding to the image. After the configuration, channel 1 is started.

Then the test set channel 2 of the DMA to perform the third command. The test programs the source registers of this channel with the starting address and the length in bytes of the alpha parameters. The destination register regarding the length is set to 0. In this way, the channel will only pick the alpha values without storing anything back. The test also enables the streaming interface of the channel. After the configuration, channel 2 is started.

When all the channels are correctly configured and enabled, the test starts to poll, for each channel, a status register that indicates if the channel has finished the given command. In this way, it can sense the completion of the three commands. After the three channels have finished, the test repeats the described operations according to the configuration parameter regarding the number of blendings to perform. Once all the blendings are performed, the test can end the simulation.

After the simulation is finished, it is possible to analyze the results by looking at the waveforms produced during the test execution. In order to mark the test successful, it is essential that all the blendings were executed correctly. This can be checked by looking

at the Stream Engine interfaces. All the four AXI4-Stream Interfaces should show the transfer of the same number of packets. It should equal the number of blendings set up in the test configuration. In addition, the DMA should have kept the flush signal of channel 0 deasserted for the entire simulation. This test does not perform any image content check for the blendings because the memory attached to the DMA is filled with zeroes at the beginning of the simulation.

This test can also be used to recreate two scenarios where the three commands needed to perform a blending are set up incorrectly due to a coding mistake. The first can be created by enabling a specific flag in the configuration phase of the test. With this flag, the test set the length in bytes of the resulting image to a lower value concerning the true one in the destination registers of channel 0. It is important to remember that the Engine will produce the correct amount of data that is superior to the one expected by the channel. According to the DMA specifications, when a channel receives from the streaming interface a number of data bytes greater than the one programmed in the registers, it raises the flush signal towards the Stream Engine. In this way, it is possible to obtain a flush case for the Engine and see how it responds in this situation. The second scenario can be created by enabling another specific flag. With this other flag, the test will set incompatible values for the length in bytes in the source registers of the three channels. In this way, it is possible to create a mismatched scenario where the Engine will see three images with a different number of pixels coming from the DMA. The Engine can detect the error and recover from this situation without stalling the AXI4-Stream interfaces.

All the described test scenarios were performed a few times on the Engine, always obtaining a successful outcome.

4.2.2 Actual blending test

The second developed test was named *Actual blending test* because it aims to perform an actual alpha blending using authentic images and alpha parameters. The images adopted for this test are the same used in the first chapter to describe the Engine capabilities. They are reported again here for better readability.

The execution of this test is divided into three main phases. The first phase consists in filling the memory contained in the testbench with the input images. It was created a Python script that reads the input images and creates a memory representation text file containing them. Some excerpts of this file are reported below:

```
00000000 ff ff ff ff ff ff ff ff
00000008 ff ff ff ff ff ff ff ff
00000010 ff ff ff ff ff ff ff ff
...
009ffff0 00 00 00 00 00 00 00 00
009ffff8 00 00 00 00 00 00 00 00
00a00000 ff 4f 5f 79 ff 50 60 7a
00a00008 ff 50 60 7a ff 4f 5f 79
00a00010 ff 52 62 7c ff 51 61 7b
00a00018 ff 52 60 7b ff 53 61 7c
```

```
...
013ffff0 00 00 00 00 00 00 00 00
013ffff8 00 00 00 00 00 00 00 00
01400000 01 01 01 01 01 01 01 01
01400008 01 01 01 01 01 01 01 01
01400010 01 01 01 01 01 01 01 01
...
```

The memory is little-endian ordered, and byte addressed. In each line of the file are reported eight bytes. The first excerpt shows the position of the foreground image (Figure 4.2) located starting from address `0x00000000`. The second excerpt shows the position of the background image (Figure 4.3) located starting from address `0x00a00000`. The third one shows the position of the alpha parameters (Figure 4.4) located starting from address `0x01400000`. The three images are placed in memory row by row starting from the top-left corner. The color format used in the memory representation for the foreground and background images is the ARGB8888. The alpha parameters instead are put down as one byte per pixel. The non-used locations of the memory were filled with zeroes.

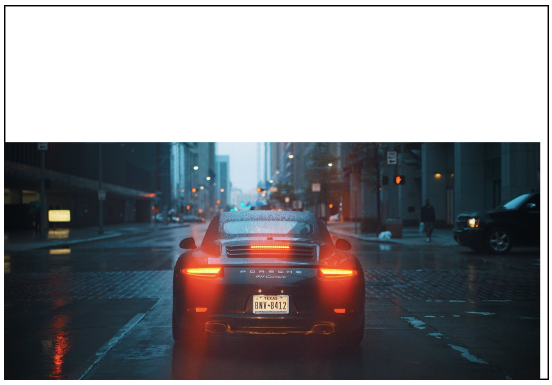

Figure 4.2: Foreground example image

Figure 4.3: Background example image

Figure 4.4: Alpha channel of the foreground image

Figure 4.5: Result example image

The second phase of this test is the logical simulation. The C file for this test is quite similar to the previous test. This time only one blending is executed. The color formats and the number of pixels of the three images are fixed and set up in the configuration part. The three images are 1920 pixels large and 1318 pixels high. For simplicity, the color format used for the output image is the same used for the foreground and background ones, ARGB8888.

Similarly, as described for the previous test, the three channels are configured with the correct addresses and lengths. The logical simulation follows the C file compilation and execution. A the end of the simulation, the content of the memory is dumped in a text file. It should contain the three input images plus the output resulting image. An excerpt of this produced file is reported below:

```
...
016ffff0 00 00 00 00 00 00 00 00
016ffff8 00 00 00 00 00 00 00 00
01700000 ff 4f 5f 79 ff 50 60 7a
01700008 ff 50 60 7a ff 4f 5f 79
01700010 ff 52 62 7c ff 51 61 7b
...
```

The excerpt shows that the resulting image is located at address `0x01700000`.

The last phase of this test consists of analyzing the dumped memory file to retrieve from it the resulting image. Another python script was written to perform this operation. It extracts the pixels from the text file and stores the obtained image. The result of this final step is the picture reported in Figure 4.5. This last test shows that the Engine can do correctly alpha blending operations.

Chapter 5

Conclusions

This chapter concludes the work by comparing the initial targets against what was achieved and an analysis of the obtained results.

The initial goal of this work was to develop a hardware accelerator to attach to a DMA for enhancing its features with the image composition through alpha blending capability. As shown in the second chapter, the one regarding the design of the Stream Engine, the goal was met. The verification results shown in chapter three demonstrated that this Engine could perform the given image elaborations correctly according to the specifications. Those results also showed that the Engine could easily recover from error situations. Since the verification of the Stream Engine reached adequate levels of coverage, there were no issues in making the Engine work together with the DMA, as presented in chapter four. This confirmed that a complete unit-level verification phase could reduce the effort needed when those units are connected to work together.

Even if it is pretty evident that the developed hardware accelerator can ease the main elaboration unit workload, one future development of this work could be a fair comparison of two systems, one with the Stream Engine and one without it. In order to be fair, the two systems should rely on the same architecture with the same components, and the same task, an image composition, should be given to them among other general-purpose tasks. The presence of the other tasks in the system is essential because the objective of this hardware accelerator is not only to make the alpha blending of two images faster but also to remove that particular kind of task from the main elaboration unit list to speed up the whole system.

Bibliography

[1] T. Porter and T. Duff, "Compositing digital images," eng, *Computer Graphics (ACM)*, vol. 18, no. 3, pp. 253–259, ISSN: 0097-8930.

[2] Arm, *Amba® 4 axi4-stream protocol*, Arm, 2010.

Acknowledgements

Before concluding this thesis, I would like to thank all the people that directly or indirectly contributed to it.

Thanks to professor Maurizio Martina. Thanks for his help and advice during those last months.

Thanks to everyone I met during my internship in Arm Hungary. Thanks for the warm welcome when I joined you in September and for being always friendly and supportive, especially in times of need.

Thanks to Deborah, who has followed me on this journey since my first interview. You were there and shared my anxiety when I was waiting for answers. You were there when I moved to Budapest. You are here with me today and share the joy of this moment. I love you.

Thanks to my family. Thanks for having always encouraged me to pursue my passions and my dreams.

Thanks to all the great people I met in Collegio in the last five years and a half. You have been and always be my second family. Thanks for all the moments we shared together. I will miss you all.