TU/e technische universiteit eindhoven

**Faculty of Electrical Engineering**
Section Design Technology (ICS/ES)

Practical Training Report

# Design and implementation of a JPEG decoder.

S. Stuijk

| | |
|---|---|
| Coach: | J. Guffens |
| Supervisor: | prof. dr. ir. J.L. van Meerbergen |
| Date: | December 2001 |

# Design and implementation of a JPEG decoder

Supervisor: prof. dr. ir. J.L. van Meerbergen
Eindhoven University of Technology
Department of Electrical Engineering
Information- and Communications Systems Group (ICS)

Coach: J. Guffens
Adelante Technologies NV
Abdijstraat 34
3001 Leuven - Belgium

Author: S. Stuijk
Id: 446407
Address: Zaanmarkstraat 26; 4811 RL; Breda
Phone: +31-76-5145396
E-mail: s.stuijk@stud.tue.nl

## Abstract

JPEG is a widely used image compression technique. It is used in image processing systems such as copiers, scanners and digital camera's. These devices often require high-speed image compression system. To fulfill this need, an IP-block that performs the JPEG decoding is used in a digital signal processor.

The report describes the operations needed to perform a valid decoding of an encoded image. The operations include decompression, inverse discrete cosine transformations, color conversion and reordering.

A multi-processor architecture is described which implements the JPEG decoder. The division of the decoder into several subsystems is discussed as well as the functional requirements for these subsystems.

To enable communication between the various subsystems a communication protocol is introduced. This protocol allows a subdivision of the system between almost all operations in the decoding process.

The report describes further the functional specification of all processors and the communication logic in the system.

A C model is presented to verify the resulting architecture. Simulations of the model show that the chosen architecture meets the requirements for a JPEG decoder as specified by the ITU.

The design is then translated into a VHDL and SystemC model using the A|RT tools. The implementations of the system in both languages are discussed. Both the VHDL and SystemC model of the JPEG decoder have been simulated. This shows that the chosen approach for a multi-processor system can easily be made with the A|RT tools. It further shows that the SystemC models generated by A|RT can be used to optimize the design in an efficient way.

The report also contains a number of recommendations for changes in the A|RT tools and the SystemC implementation of A|RT. The most important recommendations are that there is a need for good documentation about the SystemC implementation of A|RT. It is further suggested that the SystemC model of the A|RT processor should be modified to make it more understandable.

The report ends with a number of recommendations for changes in the course "Design of large scale integrated circuits" that is lectured at the TU Eindhoven. The most important recommendation given here is to include the SystemC simulation in the course. The students can then experience what cycle-accuracy means.

## Acknowledgements

During my traineeship I had the opportunity to work in an inspiring environment, which eventually resulted in this report.

I would like to thank Jan Guffens, my supervisor at Adelante Technologies. He could always find time to discuss my ideas and answer my questions. He always showed me the bottlenecks in my ideas and motivated me to search for better solutions.

I would also like to thank all people from the Software Engineering department of Adelante Technologies. They were always willing to make modification to their software to overcome the problems I experienced with the A|RT tools. Their effort allowed my to create a working solution within the time of my traineeship.

Finally, I would like to thank Jef van Meerbergen, my supervisor at the university. He offered me the opportunity to work on this very interesting and stimulating project.

# Contents

# 1. Introduction

In 1986, the CCITT, ISO, industry and universities started a standardization group, the Joint Photographic Experts Group (JPEG). The goal for this group was to develop a new image compression technique. This resulted in an official standard in the beginning of the nineties. This standard [1] describes the coding and decoding of continues-tone still images. The standard defines that a number of different coding techniques may be used. This includes both Huffman and arithmetic coding, which can both be used in differential and non-differential form. The standard defines also that for both coding techniques a number of different differential cosines transforms may be used. This has eventually resulted in fourteen different methods for coding a JPEG image.

One of these methods is the baseline JPEG coding method. We will describe the decoder used for this JPEG coding method in chapter 2. In the following chapter, we will make an analysis of the data transformations involved in the decoding process and their hardware requirements. Based on this analysis, we will present a multi-processor architecture for a system that forms the described JPEG decoder.

The processors in the system communicate with each other via a communication protocol. This protocol is described in chapter 4. The functional specification of all processors and the communication logic in the system will be given in the following chapters.

A C model was written that is based on the functional specification and architecture developed for this JPEG decoder. This C model will be discussed in chapter 8.

In chapter 9 and chapter 10, a VHDL and SystemC model for the JPEG decoder will be discussed. We will present an implementation for both models and present some experimental results that were derived using these models.

The development of the different parts of the JPEG decoder will be done with the use of the A|RT tools. These tools are developed by Adelante Technologies. They allow system designers to develop processor-like architecture interactively and "compile" an algorithm-oriented, behavioral C code description onto it.

Some conclusions and remarks about the designed JPEG decoder will be made in the last chapter. Here, we will also give recommendations for changes in the SystemC integration of A|RT. We will further give some suggestions on changes that can be made to the course "Design of large scale integrated circuits". This course is lectured at the TU Eindhoven. It uses the JPEG decoder as an example to present the top-down development of digital systems. The suggested changes to this course may help students to increase their understanding of the design process.

# 2. JPEG decoder

The JPEG decoding process is described in this chapter. First, a general overview of the decoder is given in paragraph 2.1. A more detailed description of the flow of the compressed image data is then given in the following paragraphs.

## *2.1. General overview*

A general overview of the JPEG decoding process will be given in this paragraph. For more information about the decoding process, we refer to the official JPEG standard [1].

As a starting point for our overview of the JPEG decoder, we could take the functional definition of the JPEG decoder. This definition states: "a JPEG decoder is capable of reconstructing image data from a stream of compressed image data". This requires that some transformations be applied to the compressed image data. This should result in the reconstruction of the image data.

The JPEG standard defines fourteen different methods for coding image data. In this report however, we will only use one type of coded image data. This coding method is called the baseline process. It is the basic JPEG decoding process and it is supported by all JPEG decoders that use DCT coding. The complete specification of this process can be found in Annex F of the JPEG standard [1].
In the rest of this report, we will refer to the baseline process as the JPEG coding/decoding process.

The fact that this coding method forms the basic coding method for all DCT-based JPEG decoders makes it an interesting coding/decoding method to implement. For that reason was it selected to be implemented in this project.

The JPEG decoding process is graphically depicted in Figure 2-1.

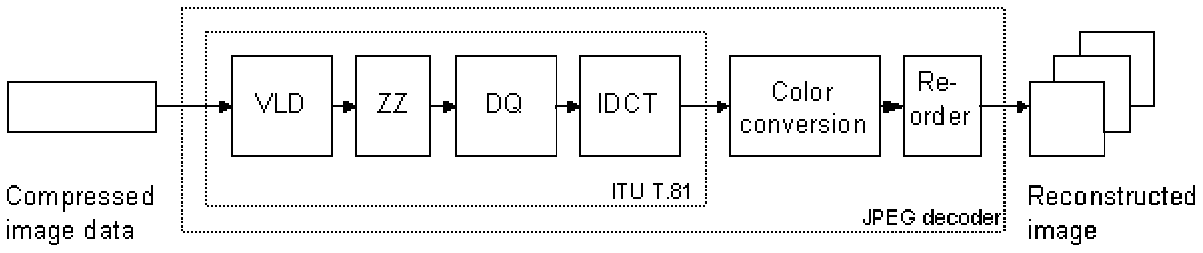


**Figure 2-1 JPEG decoder**

Before we explain the operations preformed by the decoder, we look at the encoder. The JPEG encoder divides an image in blocks of 8 by 8 pixels. The encoder then has a number of blocks, which when placed in the right order form the original image. The encoder applies a number of operations on each of these blocks. These operations include a discrete cosine transform, zigzag scan, quantization and variable length encoding. The result of these operations, and of the encoder, is a compressed image.

The decoder has to revert the transformations applied by the encoder to the image data. The decoder therefore takes the compressed image data as its input. It then subsequently applies a variable length decoding [VLD], zigzag scan [ZZ], dequantization [DQ], inverse discrete cosine transform [IDCT], a color conversion and reordering to it. It then obtains the reconstructed image.

Details about the different transformations applied to the compressed image data can be found in Appendix B.

In the above-described JPEG decoder, we applied all transformations depicted in the dashed box labeled "JPEG decoder" of Figure 2-1. The JPEG standard however requires only that the transformations present in the dashed box labeled "ITU T.81" be applied to the compressed image data. The question in now why we added the color conversion and reordering step.

The color conversion process is added to convert from the YCbCr color space to the RGB color space. The reordering step re-orders the output in a line-by-line fashion. These operations are not necessary in the decoding process and are therefore omitted in the standard. However, to facilitate the displaying of the result of our decoder, we added this color conversion and reordering steps to the JPEG decoder.

## 2.2. Data flow

In the previous paragraph, we gave an overview of the JPEG decoding process. We introduced the data flow for the compressed image data in Figure 2-1. However, we did not say how the decoder should obtain the information needed to decode the image. The decoder for instance has to know the image size and Huffman tables it has to use. Because this information is specific to the image being decoded, it has to be present in the compressed image data. We will describe in this paragraph the method used in JPEG to store this information in the compressed image data. We therefore have to take a closer look at the flow of the compressed data.

This compressed image data forms a byte stream input for the decoder. This byte stream contains so called markers. A marker is a two-byte combination, which identifies a structural part of the compressed image data. The first byte is always 0x'FF'. The second byte is defined in the JPEG standard. This byte indicates which of the structural parts of the compressed image data follows the marker in the byte stream.

The JPEG standard describes the syntax for the flow of the compressed image data. The syntax for flow of this compressed image data in a baseline JPEG decoder is given in Figure 2-2.

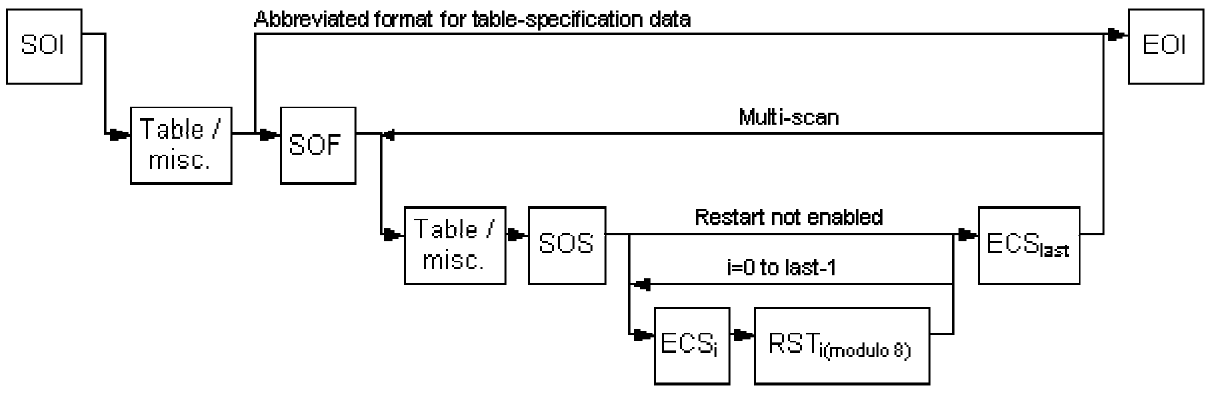


**Figure 2-2 Flow of compressed data syntax**

A valid JPEG compressed image data stream always starts with a start of image [SOI] marker. After the SOI marker a number of different markers may be found. These identify for instance quantization or Huffman tables needed for the decoding. The tables supported by the decoder are listed in Table 2-1. After zero or more of these tables, a start of frame marker [SOF] may be found. After this SOF marker, these tables can also be defined. After zero or more of these tables, a start of scan [SOS] marker must be found. After the SOS, we find a number of entropy-coded segments [ECS] in the compressed image data stream. These ECS contain the coded values for all pixels that comprise the image. These pixel values are grouped in so called minimal coded units [MCU]. They form a basic blocks for the decoder and will be discussed in detail in the following paragraph.

An ECS contains one or more MCUs. If the entropy-coded segment does not contain the last MCU of the image, then a restart marker [RST] is found after the entropy-coded segment in the compressed image data. After this restart marker, another entropy-coded segment starts. This process is repeated until all entropy-coded segment are processed by the decoder.

Then another scan may be found in the compressed image data, or the compressed image data ends with an end of image marker [EOI].

**Table 2-1 Tables/miscellaneous marker segments supported by JPEG decoder**

| Code Assignment | Symbol | Description |
|---|---|---|
| 0x ' FFDB' | DQT | Define quantization table |
| 0x ' FFC4' | DHT | Define Huffman table |
| 0x ' FFDD' | DRI | Define restart interval |
| 0x ' FFFE' | COM | Comment |
| 0x ' FFE0' | APP | Application data |

## 2.3. Minimal Coded Unit

We introduced the notion of a minimal coded unit [MCU] in the previous paragraph. However, we did not explain what a MCU is. To do this, we need to have a better understanding of the method used in the JPEG standard to encode color information in the compressed image data.

An image can be separated in a number of color components. This result in a set of grayscale images describing the tone of the colors in the image. When an image is, for instance, separated into its red, green and blue components, you obtain three grayscale images describing the red, blue and green tones in the image.
Every grayscale image describing a tone can be divided smaller parts using a grid of 8 by 8 pixels. This array of 8x8 sample values is called a block in the JPEG standard. We use the same definition for the word block in this report.

An MCU is now defined as the smallest number of blocks, which contains all samples of every component in the scan that describe a certain region of the image.
Depending on the horizontal and vertical sampling factors of every color component it may be necessary to take one or more blocks of that component into a MCU. The maximum number of blocks in a MCU is however limited by the standard to at most ten.

Figure 2-3 illustrates the definitions of a block and a minimal coded unit. The original image has been separated into the color components red, green and blue. The top-left group of 8x8 of sample values of the red color component of the image is called block 1. The top-left group of 8x8 sample values of the green color component of the image is called block 2. The same group of sample values in the blue color component of the image is called block 3. Block 1, 2 and 3 together define the values of all samples of all color components in the top-left region of the image and they therefore form a minimal coded unit (MCU 1).

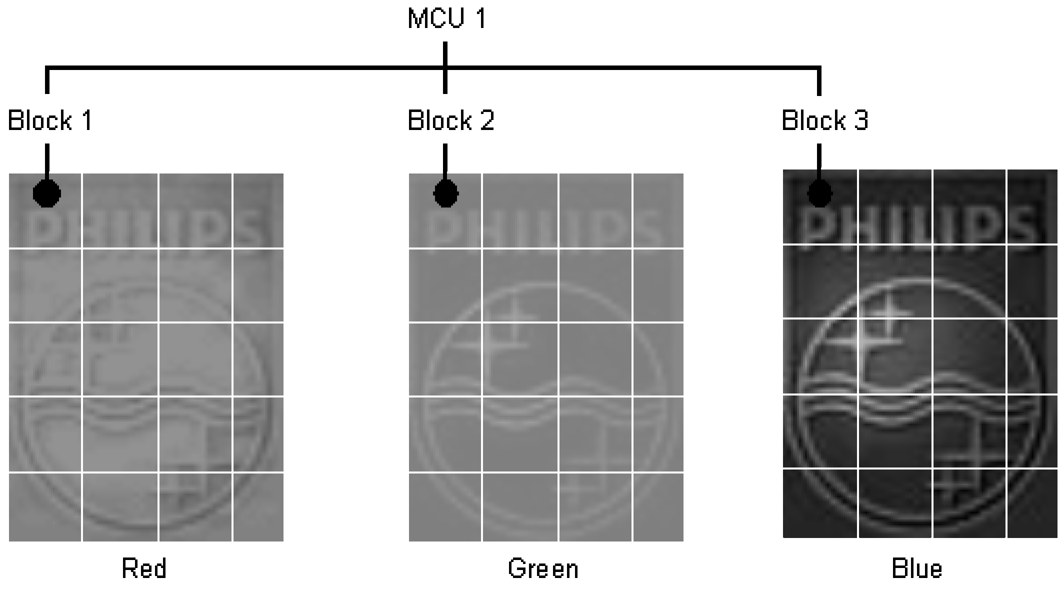


**Figure 2-3 Example of block and MCU**

In the same way, we can identify block 4 as the block next to block 1, block 5 as the block next to block 2 and block 6 as the block next to block 3. These blocks then form MCU 2. This pattern can be extended until all block of all color components are grouped into an MCU. We then have a set of MCUs that describe the complete image.

This process of creating block and grouping them into MCUs is performed by the encoder. The decoder finds these MCUs one after another in the compressed image data. The decoder can then split an MCU into its representing blocks and decode those. The order in which the decoder finds the MCUs is shown in Figure 2-4.

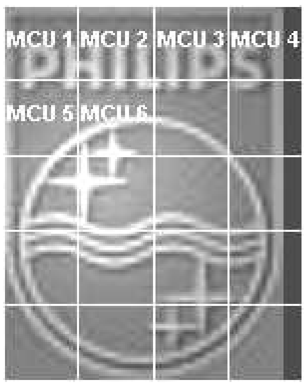


**Figure 2-4 Order of MCUs in compressed image data**

# 3. JPEG decoder architecture

A general JPEG decoder was introduced in the previous chapter. This JPEG decoder is shown in Figure 3-1. The decoder can be subdivided into five transformations: variable length decoder, zigzag scan, dequantization, inverse discrete cosine transform and color conversion.

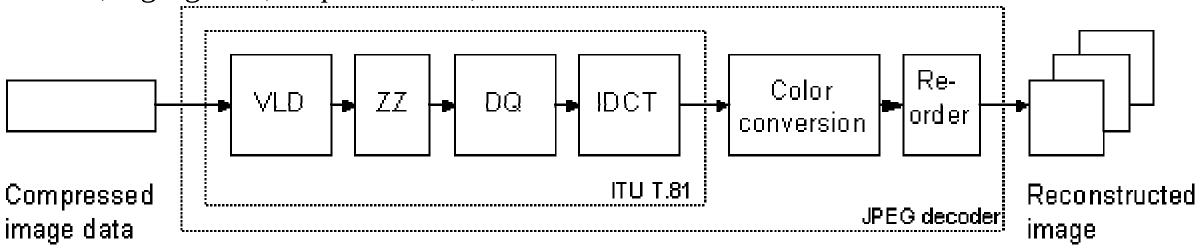


**Figure 3-1 JPEG decoder**

Our goal was to create a multi-processor system for this JPEG decoder. This means that we must group the identified subsystems in two or more groups. These groups of subsystems can then be realized as separate systems and be connected to each other to form the JPEG decoder.

The basic form of a multi-processor system is a system in which two subsystems are present. The communication, in such a multi-processor system, is limited to two subsystems and is therefore rather simple. Because the goal of this project is to implement a multi-processor system to demonstrate how this can be done using the A|RT tools, the choice was made to create a multi-processor system with two subsystems. We then realize a multi-processor system, but we can keep the communication logic relatively simple.

As can be seen in Figure 3-1, all transformations are done one after another. This implies that our multi-processor system will have two subsystems that are placed in series. This results in a multi-processor system as shown in Figure 3-2. The dashed circle forms the complete system, which is connected with the outside world via the connections one and three. The system is composed of two subsystems, which communicate with each other via connection two.

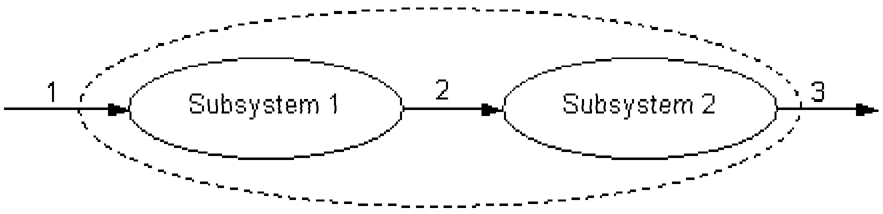


**Figure 3-2 A multi-processor system architecture**

We now have to find a match between the JPEG decoder of Figure 3-1 and the multi-processor system of Figure 3-2. If we compare these figures, we can see that the compressed image data uses connection one for our multi-processor system. The compressed image data is connected to the VLD in the JPEG decoder. Therefore, the VLD must be incorporated in the first subsystem. We can see further in the image that the re-ordering is connected to the output, which is connection number three in the multi-processor system. Therefore, the re-ordering must be present in the second subsystem.

We now have to divide the ZZ, DQ, IDCT and color conversion over the two subsystems. Let us therefore first look at the data consumption and production rate of the various parts of the system. The data consumption of the VLD is not relevant, because it receives its data from the outside world. The VLD produces however data in blocks. The zigzag scan consumes and produces one block at a time. The dequantization and IDCT also consume and produce data on a block-by-block basis. The color conversion and re-ordering requires one or more (up to ten) blocks before they can run. The color conversion however produces data in a block-by-block basis and sends this to the re-ordering unit. The data production rate of the re-ordering unit is irrelevant as this unit sends its data to the outside world.

However, all data transport between the various parts within the system is done in blocks. This implies that the communication over connection 2 of our multi-processor system of Figure 3-2 is always in blocks. This implies that every division of the JPEG decoder in two subsystems requires the same data rate over the internal communication logic. Therefore, our subdivision of the JPEG decoder does not influence the communication load of the system.

A second way to look at the division of the JPEG decoder is to look at the system load of the various parts of the JPEG decoder. This might give us information on the required computing time of the various parts of the system. We, of course, strive to a solution in which both subsystems need more or less the same computing time. This is because subsystem two can start as soon as subsystem one has produced one block. After subsystem one has produced its last block, we want subsystem two to finish as quickly as possible. This implies that it must need around the same computing time for his operations as subsystem one needs for his operations. Therefore, a match in system load is necessary.

In Table 3-1, we listed the system load for the various parts of our JPEG decoder. These system loads are valid based on the hardware requirements when a standard processor is used. This means a processor in which only standard function blocks (e.g. ALU, ACU etc.) are used. There is only one exception. This involves the bit handling by the variable length decoding. If this is done on a standard processor, the VLD will consume much more of the system load. The figure for the VLD is therefore not based on the system load on a standard processor but on a system in which the bit handling is done more efficiently.

**Table 3-1 System load for parts of JPEG decoder**

| Part | System load [% of total load] |
|---|---|
| VLD | 35 |
| ZZ | 5 |
| DQ | 10 |
| IDCT | 20 |
| Color conversion | 15 |
| Re-ordering | 15 |

As mentioned before, we strive to a division in which the system load of the two subsystems is both half of the overall system load. Using this requirement and Table 3-1, we are now able to make a division. Subsystem 1 will incorporate the VLD, ZZ and DQ. Subsystem 2 will incorporate the IDCT, color conversion and re-ordering.

This division leads to the JPEG decoder architecture shown in Figure 3-3. The first processor, bit stream processor, realizes subsystem one. The second processor, image processor, realizes subsystem two. The second processor is also responsible for the output of the image, while the first processor does the input handling. Both processors communicate with each other via a communication channel in the communication logic.

The image processor however will require access to some image data such as the image size and number of color components in the image. This information is produced in the bit stream processor. To give the image processor access to this data, the data should be stored in a memory that is embedded in the communication logic and can be accessed by both processors.

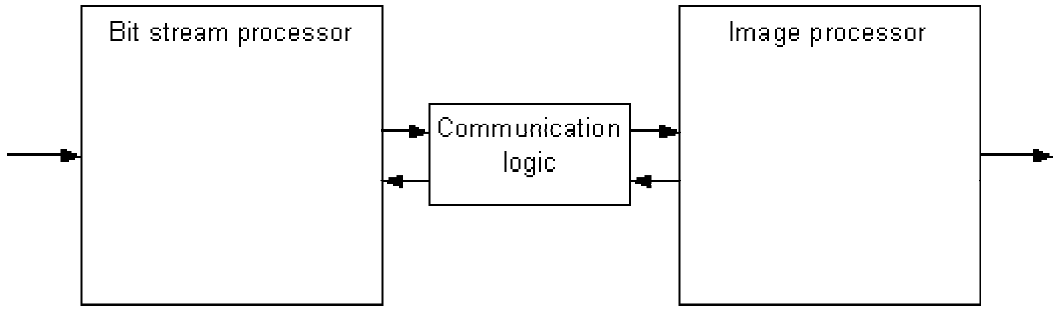


**Figure 3-3 JPEG decoder architecture (1)**

When we presented the system load for the various parts of our JPEG decoder in Table 3-1, we mentioned one constraint. This was that the bit handling in the variable length decoder was done in an efficient way.

When we would use a standard architecture to implement this bit handling, many shift and logic operations would be required on an ALU. This would have a serious impact on the systems performance. Therefore, an application specific unit [ASU] is required to do the bit handling. We will call this ASU the bit ASU.

The compressed image data consists mainly of data used in the variable length decoder. Only a small part contains tables and other image information. However, most of the data from the input is needed by the bit handling routine of the variable length decoder. Therefore would it be logical to incorporate the input into this ASU. The bit ASU is then responsible for providing access to the bits and bytes of the compressed image data.

When we make these changes to the JPEG decoder architecture of Figure 3-3, we obtain the architecture shown in Figure 3-4 for our JPEG decoder.

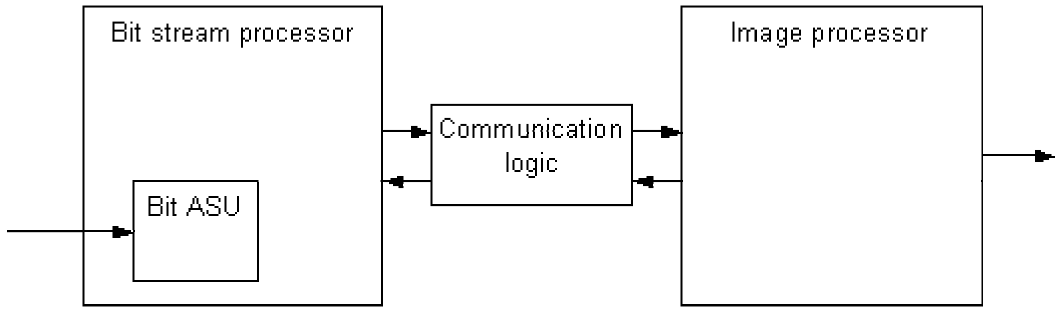


**Figure 3-4 JPEG decoder architecture (2)**

We now have a multi-processor architecture, which implements the JPEG decoder of Figure 3-1.

# 4. Communication protocol

In the previous chapter, we introduced a multi-processor architecture that realizes the JPEG decoder of chapter 2. The processors in the system communicate with each other via a communication protocol on the communication channel in the communication logic. This chapter describes the communication protocol used by the two processors to communicate with each other over this communication channel.

## 4.1. Communication channel

The bit stream processor and image processor communicate with each other via the communication logic, as can be seen in Figure 3-4. The bit stream processor is sending blocks to the image processor using the communication channel. The image processor does not have to send data back to the bit stream processor. Therefore, a one-way communication channel will be sufficient. We now define that in the communication protocol there is no acknowledgement between the two processors that the data has been received. The communication channel however, can indicate that it does not accept or cannot deliver the data. The processor that want to read from or write to the communications channel has to hold its operations and has to wait until the packet is accept or can be delivered when this happens.

## 4.2. Packets

Communication in the communication channel is done via so called communication packets. If the bit stream processor has data for the image processor, it sends a packet to the communications channel. The packet is stored there until the image processor asks for a new packet. Then the oldest packet in the channel (packet longest in channel) is then send to the image processor.
We will define in this paragraph the type of packets that can be sent via the communication channel. We will also define the format for those packets.

### 4.2.1. Type of packets

We have to look at the bit stream processor to identify what information it wants to send to the image processor. Based on that, we can define different communication packet types that allow the bit stream processor to send this information to the image processor.

The first block of information the bit stream processor wants to send is a block that has been dequantized. Let us define the packet type needed to send this block as a *block packet*.
After the bit stream processor has found the end of image marker, it has to inform the image processor that it has sent all block packets. This can be done via an *EOI packet*.
The exact format of a packet will be defined in the next section. Before that, we will take a closer look at the two different packet types.

**Block packet**
The first packet we defined was the block packet. Let us now take a closer look to what information should be packaged in this packet.

The image processor should know where a certain block belongs in the image. First, it must know to which color component a block belongs. It can then select the correct color component image.
After the image processor has selected the correct color component image, it must know the position of the block in the color component image indicated along two axes, a horizontal and

vertical ax. The vertical ax is indicated by the MCU row. The horizontal ax is indicated by the MCU column.
This implies that three parameters are added to every block that is send from the bit stream processor to the image processor over the communication channel.
The bit size of the elements in the block is 11 bits. The values of the MCU row, MCU column and component identifier are all smaller then 11 bits. Therefore, the communications channel should be at least 11 bits wide.

**EOI packet**
The second packet type is the EOI image packet. This packet only has to inform the image processor that the bit stream processor has put all block onto the communications channel and that it has found the end of image marker in the compressed image data.

### 4.2.2. Packet format
In this section, we define the format for all packet types defined in the previous section. Let us start with defining a general layout for all packet types.

A communication packet consists of communication packet elements. The width of a communication packet element is defined to be 12 bits. A communication packet is then a collection of these communication packet elements. Every communication packet must start with a communication element packet that contains a marker. This marker identifies the type of packet. The type of packet then determines the length of the communication packet (number of communication packet elements in communication packet). All valid types of packets and their markers and length are printed in Table 4-1. The general format of a communication packet is shown in Figure 4-1.

**Table 4-1 Type of packets**

| Marker | Length | Type of packet |
|---|---|---|
| 0x' 800' | 68 | Block packet |
| 0x' FFF' | 1 | EOI packet |

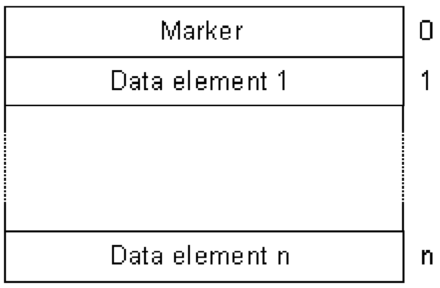


**Figure 4-1 Communication packet**

The format for the Block packet is shown in Figure 4-2 and the format for the EOI packet are shown in Figure 4-3.

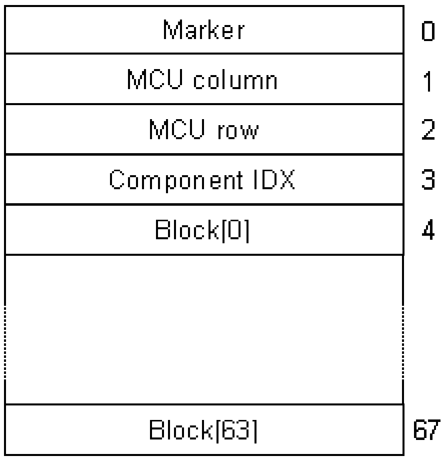


**Figure 4-2 Block packet**

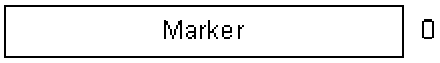


**Figure 4-3 EOI packet**

### 4.2.3. Sending a packet to the communications channel

When a packet must be sent over the communications channel, the communication packet element containing the marker must be send first. The address for this communication packet element is zero. This indicates that it is the first communication packet element of a communication packet. The other elements can be sent in any order. The order in which they should be placed inside the packet must be indicated by the address that is send along with the data to the communications channel.

### 4.2.4. Receiving a packet from the communications channel

When a processor wants to read a packet from the communications channel, it should request a read operation to the channel with address zero. The communications channel will then return the first communication packet element from the oldest communication packet available in the communication channel. After reception of the communication packet element that contains the marker, the processor may request the other communication packet elements in the communication packet in any order. This is done by sending the address of the desired communication packet element to the communication channel. The communication channel will then return the requested element.

# 5. Bit stream processor

In chapter 3, we presented an architecture that implements the JPEG decoder of chapter 2. Part of this architecture is the bit stream processor. We will give a functional description of the various parts of the bit stream processor in this chapter.

In chapter 3 was defined that the bit stream processor is responsible for the variable length decoding, zigzag scan and dequantization. It should therefore contain functional units that are capable of performing these operations.

The bit stream processor should further communicate with the communication channel. This is done via the output. This unit sends communication channel packets to the communication channel.

The architecture defined that all access to compressed image data should be done via a unit called bit ASU. This bit ASU implements a number of functions to facilitate the access to the compressed image data. The bit ASU uses a unit called input block to receive portions of the compressed image data.

The bit stream processor further contains a unit called control logic. This unit controls the data flow in the processor.

The units present in the image processor and their connections are shown in the following figure.

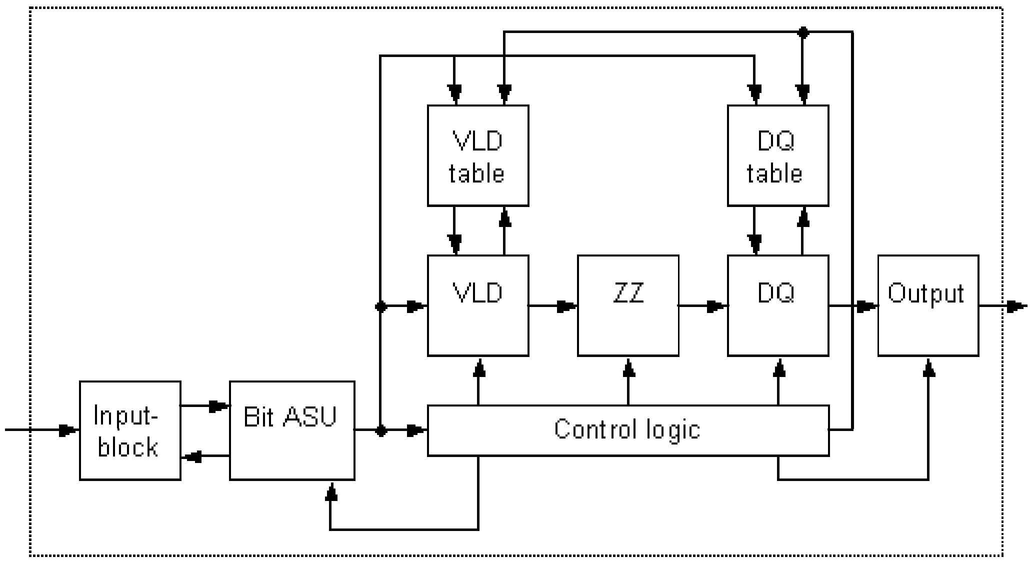


**Figure 5-1 Functional units in bit stream processor**

A functional description of the control logic, VLD, ZZ, DQ, Bit ASU and input block unit will be given in the following paragraphs.

## *5.1. Control logic*

The control logic is responsible for controlling the flow of the decoding process. This means that it must request markers from the bit ASU. Upon reception of a marker, it should perform the needed action to process the found marker. The actions taken by the control logic when it finds a specific marker are listed in Table 5-1.

**Table 5-1 Actions taken by control logic upon reception of marker**

| Marker | Action |
|---|---|
| SOI | Allow decoding of image. |
| EOI | Stop decoding of image. |
| APP | Read segment from input stream. |
| COM | Read segment from input stream. |
| DRI | Set restart interval. |
| DQT | Load dequantization table. |
| DHT | Load Huffman table. |
| SOF | Process frame header. |
| SOS | Process start of scan header;<br>Process all MCUs in compressed image data;<br>Send EOI marker over communications channel. |

## 5.2. VLD unit

The VLD unit should perform the variable length decoding operation. The operation of the variable length decoder is given in appendix B.4.

The JPEG standard allows the use of two DC tables and two AC tables for a baseline sequential decoder. The VLD should decide which table is needed for the decompression.

The tables needed by the VLD unit can be found in the VLD table. The control logic is responsible for loading the correct Huffman tables in the VLD table. The VLD unit assumes that the correct tables have been loaded.

The VLD requires bits from the compressed image data in order to function. To get one or more bits from the compressed image data, the VLD can ask the bit ASU for these bits.

## 5.3. ZZ unit

The ZZ unit should perform the zigzag scan operation. The mathematical definition of this operation is given in appendix B.3.

## 5.4. DQ unit

The DQ unit should perform the dequantization. The mathematical definition of this operation is given in appendix B.2.

The JPEG standard allows the use of up to four dequantization tables for a baseline sequential decoder. To select the correct dequantization table, an extra parameter should be supplied to the dequantization unit.

The dequantization tables needed by the dequantization unit can be found in the DQ table. The control logic is responsible for loading the correct dequantization tables in the DQ table. The dequantization unit assumes that the correct tables have been loaded.

## 5.5. Bit ASU

The compressed image data can be modeled as a stream of bytes. The JPEG decoder moves trough this byte stream while it decodes the image. The bit ASU is used to facilitate the JPEG decoder in accessing these bits and bytes. The bit ASU has therefore a number of procedures. These procedures and the interface of the bit ASU are described in this paragraph.

### 5.5.1. Interface of bit ASU

The interface of the bit ASU is shown in Figure 5-2.

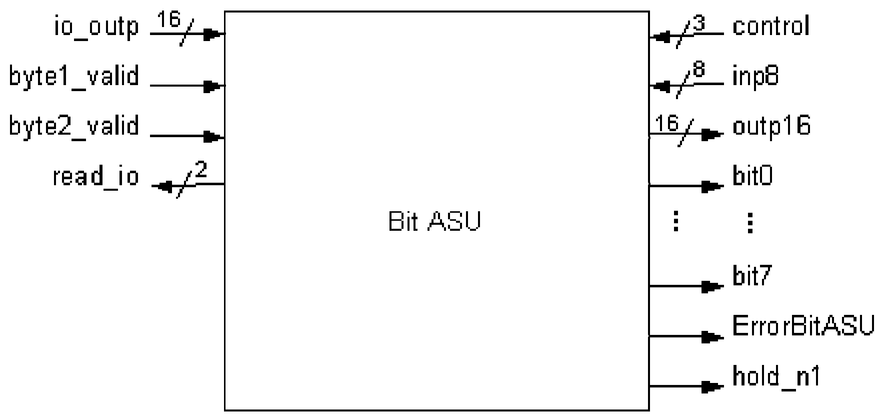


**Figure 5-2 Bit ASU interface**

It receives a 16-bit vector on its *io_outp* port. This vector contains two bytes with the orientation shown in Figure 5-3. The ports *byte1_valid* and *byte2_valid* indicate whether these bytes contain valid data.

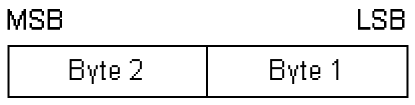


**Figure 5-3 Format of bitvector io_outp**

The bit ASU must indicate on its *read_io* port whether it has used zero, one or two valid bytes from the *io_outp* vector. This is done by setting the *read_io* port to zero (0b' 00' ), one (0b' 01' ) or two (0b' 10' ).

The selection of the appropriate procedure is done via the control input. The values for the different procedures may be chosen by the implementation.

The input port *inp8* is used to supply the bit ASU with parameters required by some functions. The output ports *outp16* and *bit0, bit1, bit2, bit3, bit4, bit5, bit6* and *bit7* are used to output the results of the different procedures.

The *ErrorBitASU* port is normally zero. In case an error occurred during the execution of the unit, the *ErrorBitASU* is set to one.

It is possible that the bit ASU needs more bytes from the byte stream then there are available at the *io_outp*. It then set the *hold_n1* output to one. The processor must then hold its operations until the *hold_n1* output of the bit ASU goes back to zero. At that time, the requested procedure has finished its execution and the result is available at the output.

### 5.5.2. ReadMarker procedure

This procedure searches for the next marker in the remaining compressed image data stream.

When the procedure finds the marker, it splits the marker into its representing bits. These bits are returned via the ports *bit0* through *bit7*.

The process that called the ReadMarker procedure obtains a set of flags that represent the marker. The calling procedure can use these flags to make a control decision.

A marker is assumed to be found if the first of two bytes equals 0x' FF' and the second byte is not 0x' FF' or 0x' 00' .

**5.5.3. ReadSegmentSize procedure**
This procedure assumes that the next two bytes in the compressed image data stream contain the size of the segment that is being read. It therefore reads the next two bytes from the stream. It then has the size of the segment in bytes. This includes the two bytes that contain the length. It then subtracts a value of two from it and returns the resulting value via the *outp16* port. This equals then the number of bytes remaining in the segment being read.

**5.5.4. ReadByte procedure**
The ReadByte procedure reads one byte from the compressed image data stream and returns it via the *outp16* port. The byte is placed in the LSB side of the *outp16* bit vector.

**5.5.5. GetBits procedure**
This procedure reads the number of bits requested via the *inp8* port from the compressed image data and returns these bits via the *outp16* port. The bits are located in the LSB side of the *outp16* bit vector.
It is possible that a byte from the compressed image data is not requested completely. The GetBits procedure should then remember which bits of this byte have not been outputted. Those bits must be used first to compute the output when the GetBits procedure is called again.

**5.5.6. SkipUntilByteBoundary procedure**
After this procedure is called, the GetBits procedure will not return the bits of a partially outputted byte. In turn, the GetBits procedure will start with a new byte from the compressed image data.

## *5.6. Input block*

The compressed image data can be modeled as a stream of bytes. The JPEG decoder moves trough this byte stream while it decodes the image. Therefore, the input block outputs a bit vector (*outp16*) of two bytes length. This bit vector contains zero, one or two bytes from the byte stream that have not yet been read by the JPEG decoder and follow directly the last read byte. The input block also outputs two Booleans (*valid_byte1* and *valid_byte2*) that indicate whether zero, one or two of the bytes in the bit vector *outp16* are valid. In turn, the bit ASU returns the number of bytes (zero, one or two) it has read from the byte stream.

The input block is responsible for supplying the bit ASU with the next two bytes in the byte stream. It therefore has a circular buffer of two bytes. The input block must strive to fill this buffer with the last two unread bytes from the byte stream.

Every time the input block is executed, the input block checks whether it has two valid bytes. If that is not the case, it requests a new byte from the compressed image data byte stream. This byte will then be delivered to the input block at the *byte* port. To indicate that the byte is valid, the *valid* port of the input block must be one.

The functional behavior of the input block can be described by the state transition diagram of Figure 5-4. The state transition functions that correspond to the transitions in this figure are shown in Table 5-2.

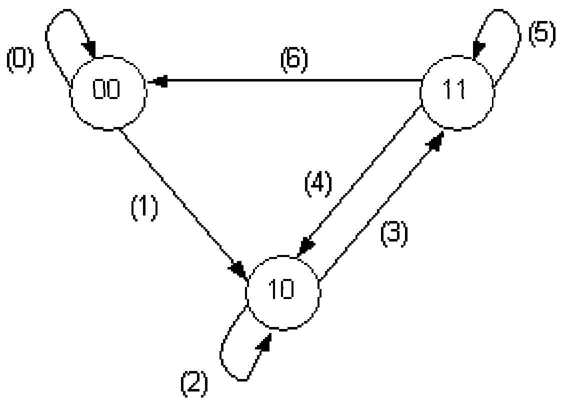


**Figure 5-4 State transition diagram for input block**

**Table 5-2 State transition table for input block**

| Transition | Input | | Output | | |
|---|---|---|---|---|---|
| | read_io | valid | byte1_nxt | byte2_nxt | req |
| (0) | 0 | 0 | - | - | 1 |
| (1) | 0 | 1 | byte | - | 1 |
| (2) | 0 | 0 | byte | - | 1 |
| | 1 | 1 | byte | - | 1 |
| (3) | 0 | 1 | byte1 | byte | 0 |
| (4) | 1 | 0 | byte2 | - | 1 |
| (5) | 0 | 0 | byte1 | byte2 | 0 |
| (6) | 2 | 0 | - | - | 1 |

The state *'00'* corresponds to byte 1 and byte 2 invalid. The state *'10'* corresponds to byte 1 valid and byte 2 invalid. The state *'11'* corresponds to byte 1 and byte 2 valid.

The orientation of byte 1 and 2 in the outp16 bit vector is shown in Figure 5-5.

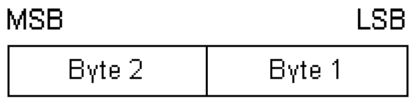


**Figure 5-5 Output format (outp16) of input block**

The interface of the input block is shown in Figure 5-6.

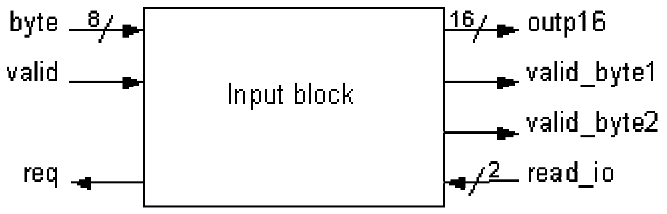


**Figure 5-6 Input block IO**

# 6. Image processor

In chapter 3, we presented an architecture that implements the JPEG decoder of chapter 2. Part of this architecture is the image processor. We will give a functional description of the various parts of the image processor in this chapter.

The image processor is responsible for the IDCT, color conversion and re-ordering. It should therefore contain functional units that are capable of performing these operations. The color conversion and re-ordering require up to ten blocks before they can execute. The IDCT unit can be executed after the reception of one block. Therefore, a storage memory between the IDCT and color conversion unit is needed.

The image processor further contains a unit called control logic. This unit controls the data flow in the processor and request communication packets from the communication channel and send the enclosed data to the correct units.

The units present in the image processor and their connections are shown in the following figure.

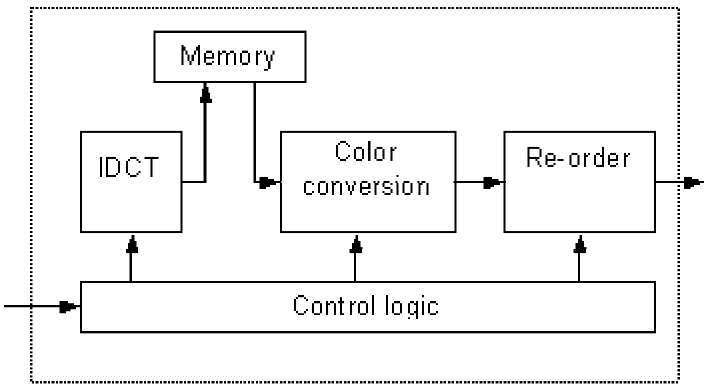


**Figure 6-1 Functional units in the image processor**

A functional description of the IDCT, color conversion, re-order unit and control logic will be given in the following paragraphs.

## 6.1. IDCT unit

The IDCT unit in the image processor should realize the inverse discrete cosine transform. It therefore takes one block as its input. It then applies an inverse discrete transformation with an 8-bit precision to it. The mathematical definition of the IDCT is given in appendix B.1.

After computation of the IDCT, the signed output samples are level-shifted. This level shifting converts the output to an unsigned representation. For 8-bit precision, the level shift is performed by adding 128 to every element of the block that came out of the IDCT. If necessary, the output samples shall be clamped to stay within the range appropriate for the precision (0 to 255 for 8-bit precision).

## 6.2. Color conversion unit

The color conversion unit in the image processor should realize the color conversion from the YCbCr coloring scheme to the RGB coloring scheme. This is only necessary if the number of

image components equals three. In all other cases, no changes to the coloring scheme will be made by the color conversion unit.
The YCbCr to RGB color conversion should be done as defined in CCIR Recommendation 601. The mathematical definition of this conversion is given in appendix B.5.

## 6.3. Re-ordering unit

The output of the color conversion unit is a minimal coded unit [MCU]. It describes all color components in a region of the image. The first MCU outputted describes the top-left most region of the image. The MCU are then outputted in a row-by-row fashion. This is shown in Figure 6-2.

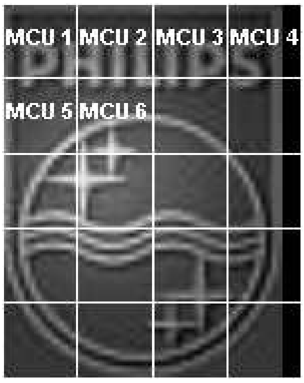


**Figure 6-2 Encoded image**

Most image display systems (e.g. video and computer) describe an image in a line-by-line manner. In this system, the first element that is outputted the value of the first color component of the first pixel of the first line. The second element outputted is the value of the second color component of the first pixel of the first line. This outputting is repeated until the value of all color components of the first pixel are outputted. The system then outputs the values of all color components of the second pixel on the line in the same way. This is then done for all pixels on that line. When the line is finished, the system repeats this procedure for the next line, until the whole image has been outputted.

The re-ordering unit should provide a conversion between the system in which MCUs are used to describe a region of the image and the system in which the image is described in a line-by-line style.

If necessary, the re-ordering unit should resize a MCU to make sure that the original image size is maintained.

## 6.4. Control logic

This unit controls the data flow in the processor and request communication packets from the communication channel and send the enclosed data to the correct units. In case, the communication packet is a Block packet, the enclosed block must be sent to the IDCT unit. When the IDCT unit has finished, the control logic has to check whether a complete MCU is present in the memory and the MCU can be processed.
When the control logic receives an EOI packet from the communication channel, it should finish its operation.

# 7. Communication logic

In chapter 3, we presented an architecture that implements the JPEG decoder of chapter 2. Part of this architecture is the communication logic. We will give a functional description of the various parts of the communication logic in this chapter.

The communication logic is responsible for the communication and data sharing between the two processors in the JPEG decoder.  The first, communication between the two processors, is realized through the communication channel. The second, data sharing, is realized through the communication RAM arbiter. A functional description of both units will be given in the following paragraphs.

## *7.1. Communication channel*

The first unit in the communication logic is the communication channel. It has to implement the communications protocol defined in chapter 4.

Based on this protocol we can describe the function of the communication channel as follows. The sender puts communication packets on the communication channel as described in paragraph 4.2.3. The data should be available at the *dout_out* port of the communication channel. The address for the data should be available at the *adout_out* port of the communication channel. A write request can be indicated by setting the *dready_out* port to one. The communication channel then stores this communication packet element in a FIFO queue. If the communications channel receives the first communication packet element, it has to check whether there is enough space to store the whole packet on the queue. If this is the case, the communication channel will accept this communication packet element, else it will reject it. Acceptance is indicated by a one at the *daccept_out* port of the communication channel. If the communication packet is rejected, this port will be zero.

The receiver can receive communication packets from the communication channel as described in paragraph 4.2.4. The address of the desired communication packet element of the current communication packet being received should be present at the *adout_in* port of the communication channel. A data request must further be indicated by a one on the *dreq_in* port of the communication channel. The data will then be made available at the *d_in* port of the communication channel. If the data is available in the communication channel, the *davail_in* port of the communication channel will be one. In the latter case, it will be zero.

The interface of the communications channel is shown in Figure 7-1.

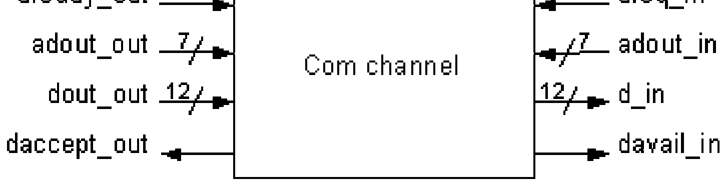


**Figure 7-1 Communication channel IO**

## *7.2. Communication RAM arbiter*

The second unit in the communication logic is the communication RAM arbiter. It implements the sharing of image data between the two processors. Both processors must therefore be able to read from and write to this RAM all data that is relevant for the decoding of the image and that is not send over the communication channel.

The functional description of the communications RAM arbiter is as follows. In case of a write event, the data present at the data input (*d*) is directly written to the memory location indicated by the address input port (*a*). The communications RAM arbiter while then return this data via its data output port (*dout*). In case of a read event, the data will be made available directly after the read request is received. The address will be read from the address port (*a*) and the data is available on the data output port (*dout*).

The communications RAM arbiter has two of the above-defined interfaces. They are labeled *_port_1* and *_port_2*. The ports can access the memory simultaneously.
The interface of the communication RAM arbiter is shown in Figure 7-2.

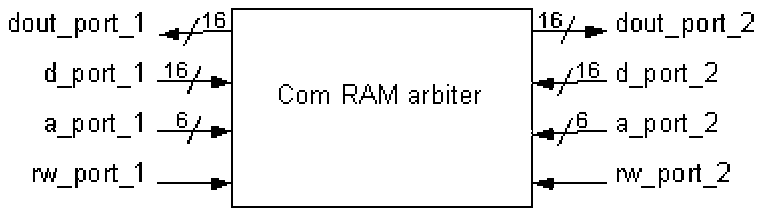


**Figure 7-2 Communication RAM arbiter IO**

# 8. C model

In chapter 3, we introduced an architecture to implement the JPEG decoder in a multi-processor system. This system includes two processors that communicate with each other via a communication channel. The processors can also share some information with each other via a shared memory that is embedded in the communication logic. A functional description of these processors and the communication logic was given in chapter 5, chapter 6 and chapter 7.

In this chapter, a C model is presented that implements the functional behavior of the JPEG decoder. The implementation of the various parts of the JPEG decoder are described in paragraph 8.1. A set of test benches is described in paragraph 8.2. They can be used to verify the behavior of the subsystems of the JPEG decoder and the JPEG decoder as a whole.

## *8.1. Implementation*

This paragraph describes the C model that implements the JPEG decoder. The complete C model can be found in the deliverable of the JPEG decoder project under the directory src/jpeg_pc2.

### 8.1.1. Communication logic

The C model for the communication logic can be found in the files com.cxx and com.h. The communication logic consists of a shared memory and a communication channel. The implementation of both units will be described one after another in the following section.

#### Communication RAM arbiter

The interface of the communication RAM arbiter is shown in Figure 7-2. This interface is implemented using the following C function header.

```
void com_ram_rw(const Uint<1> rw_port_1,
                const Uint<6> a_port_1,
                const Uint<16> d_port_1,
                Uint<16> &dout_port_1,
                const Uint<1> rw_port_2,
                const Uint<6> a_port_2,
                const Uint<16> d_port_2,
                Uint<16> &dout_port_2);
```

**Figure 8-1 com_ram_rw function**

The communication RAM arbiter must implement a dual port memory as specified in the functional description of the unit (see page 32). In C, this can be done with a standard array that is accessed twice, once for port 1 and once for port 2. The data can then be read from and written to the variables in the function header as specified.

The C model of the JPEG decoder will be a sequential model, because the C language is sequential. Due to this property, we can simplify the interface for reading and writing of a single processor to the array. It is namely so that only one processor at a time will run. Therefore, both processors can use the same port to access the memory.

In the C code, a distinction can be made between read and write events. To facilitate this reading and writing to functions are introduced that perform the read and write operation on the memory.  Their function definitions are listed below:

```
Uint<16> com_ram_write(const Uint<6> a, const Uint<16> d);
Uint<16> com_ram_read(const Uint<6> a);
```

**Figure 8-2 Read/write interface functions**

These functions are made external, so that they can be used by the two processors. The processors then only have to supply the correct address and in the case of a write event the correct address and data to access the shared memory.

**Communication channel**

The interface of the communication RAM arbiter is shown in Figure 7-1. This interface is implemented using the following C function header.

```
void com_buffer(const Uint<1> dready_out,
                const Uint<12> adout_out,
                const Uint<12> dout_out,
                Uint<1> &daccept_out,
                const Uint<1> dreq_in,
                Uint<12> &d_in,
                const Uint<12> adout_in,
                Uint<1> &davail_in);
```

**Figure 8-3 com_buffer function**

This function accepts one channel packet element at a time. This channel packet element must be stored at the right position in the channel packet that is being stored in the communication channel. The communication packet is then stored in a FIFO queue. The C model of our communication channel uses a standard FIFO queue to read and write the communication packets to. The implementation choices that are further made to implement this FIFO and the communication channel are described below.

The first implementation choice to be made has to do with a lack of specification in the functional behavior of the communication channel. The functional description does not define whether it is allowed to read from a partially send package. In the implementation, the choice has been made to forbid this. Else, the communication channel would have to administrate which elements of a packet are sent and which are not send. This makes the communication channel very complex. Therefore reading of partially send packages is not supported by the implementation. In case data from a partially send package is requested, the communication channel will say that this data is not available at the time.

The second implementation choice has to deal with a problem related to the different types of packets that can be sent over the channel. The size of a communication packet depends on this type of packet. A standard FIFO queue however assumes that every element (in our case a complete communication packet) in the queue has the same size. This is not the case in our situation. This problem can be solved using the following implementation.
Every communication packet is stored in a communication block. These blocks have the same size as the largest communication packet (68 elements). These communication blocks are then stored in the FIFO queue.
After reception of the marker (first element to be sent) the communication channel knows the number of communication packet elements that must be received before the complete communication packet is received and the communication block can be added to the FIFO.
This can be implemented using a simple if statement that checks if a marker is received (address equals zero). If a marker is received another if statement can decide whether it is an block marker and 67 more communication packet elements must be received before the

communication block is finished, or that the marker is an EOI marker and thus the communication packet has completely been send and thus the communication block can be added to the FIFO.

The interface of the *com_buffer* function is rather complex. It contains the interface for the read side and write side of the channel and it contains variables for indicating and checking whether the data must be read or written from a given address. The C model of both processors however know exactly when the want to read or write an element to the channel. Moreover, such a request must always be accepted by the channel. Therefore the ready and accept variables can be omitted.
In addition, writing to the channel using the *com_buffer* function interface requires a large number of function calls. With each function call data and an address must be supplied. To reduce this number of function calls the following solution has been implemented. The processors can write a complete communication packet into an array. When the have finished the communication packet, the can call a function *com_write_fifo* with the array as an argument. This function then writes the whole array to the FIFO using the *com_buffer* function.

The implementation of the write interface for the communication channel is then given by the following C function definition:

```
void com_write_fifo(const Uint<12> ComOut[68]);
```

**Figure 8-4 Write interface function**

For reading from the FIFO, the same solution can be used. A function *com_read_fifo* is called with an array. In this array a complete communication packet is placed. The calling function can then read the communication packet and its elements out of the array. The implementation of the read interface for the communication channel is then given by the following C function definition:

```
void com_read_fifo(Uint<12> ComOut[68]);
```

**Figure 8-5 Read interface function**

### 8.1.2. Bit stream processor
The functional behavior of the bit stream processor has been described in chapter 5. This was done by describing the functional units and their relationship. We will explain the C models used in our project for these units in this section.

The C model for the bit stream processor, excluding the bit ASU and input block, can be found in the files bitsp.cxx and bitsp.h. The C model for the bit ASU can be found in the files bitasu.h, bitasu.cxx and bitasu_impl.cxx. The C model for the input block can be found in the files ioasu.h and ioasu.cxx.

#### VLD table / QT table
The bit stream processor contains two tables, the VLD table and QT table. These tables are used to store the Huffman decoding tables and the dequantization tables. They are realized in the C model as the arrays *HTable* and *QTable*.

*HTable* can contain two DC and two AC tables. Reading and writing to these tables can be done via the following two functions.

```
void WriteHuffmanTable(const Uint<2> id,
                       const Uint<1> ACDCClass,
                       const Int<16> addr,
                       const Uint<16> value);

Uint<16> ReadHuffmanTable(const Uint<2> id,
                          const Uint<1> ACDCClass,
                          const Int<16> addr);
```

**Figure 8-6 Read/write interface functions of HTable**

The variable *id* is the table number (one or two). The variable *ACDCClass* indicates whether the DC table or AC table should be used. This variable can be set to AC_CLASS or DC_CLASS. The variable *addr* contains the position in the selected table for the data being read or written. The *value* variable then contains the data that should be written to the selected table at the selected position.

The *QTable* array can contain four quantization tables. Writing to the quantization table can be done using the following statement:

```
QTable[id * MAX_TABLE_SIZE + addr] = value;
```

**Figure 8-7 Writing to QTable**

The variable *id* indicates the quantization table that should be used. Values allowed for *id* are 0, 1, 2 and 3. MAX_TABLE_SIZE is defined to the size of one quantization table. The variable *addr* contains the position in the selected table for the data being written. The variable *value* should contain the data being written to the table.

Reading from the quantization table can de done using the following statement:

```
value = QTable[id * MAX_TABLE_SIZE + addr];
```

**Figure 8-8 Reading from QTable**

The variable *value* now contains the data at position *addr* of the quantization table indicated by *id*.

**Output**
The bit stream processor uses the communication logic to communicate with the image processor. Section 8.1.1 describes the communication interface that should be used by the bit stream processor. In the implementation of the communication channel is suggested that an array should be used to store a communication packet and then send this packet via the function *com_fifo_write*. For this purpose, an array *ComOut* is available in the bit stream processor main function *bitsp*. This array is passed to all functions that need to write to the communication channel. A function that wants to write a communication packet to the communication channel can then simple put its data in the *ComOut* array. If the whole packet is stored in the *ComOut* array, the function calls the *com_write_fifo* function to send the packet to the communication channel.

**Control logic**
The control logic of the bit stream processor is implemented in the *bitsp* function. This function calls the bit ASU function ReadMarker to retrieve image marker from the compressed image data stream. Based on the returned marker, the *bitsp* function selects via an if-else construction the correct action based on the found marker. These actions are listed in

Table 5-1. We won' t discuss the implementation of all actions here, because most of them are not very complex. The only real complex action involves the case that a start of scan marker is found. Therefore, we will discuss the action taken after a start of scan marker [SOS] was found in the compressed image data in detail.

In Figure 2-2, we showed the data flow syntax for the compressed image data. If we are processing the start of scan marker, we have to follow a part of this syntax. The part of the data flow syntax that must be used when executing the action based on the start of scan marker is shown in Figure 8-9.

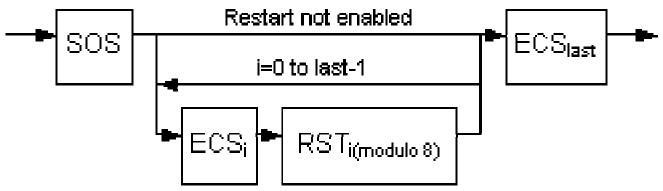


**Figure 8-9 Data flow when processing SOS marker**

First, the start of scan header, following the start of scan marker, is processed. This is done in the *ProcessScanHeader* function. After that, the number of restart is calculated. This number of restarts plus one equals the number of entropy-coded segment in the scan. The number of found restarts is then set to zero, as no restarts have been found yet.
The number of MCUs contained in one entropy-coded segment is also calculated.

Then the prediction of the Huffman decoder and the bits stored in the bit ASU for the GetBits procedure must be cleared. This is done via the functions *reset_prediction* and *skip_until_byte_boundary*.

The processing of the SOS marker has now advanced to the point at which the first entropy-coded segment can be processed. This is done by calling the *ProcessMCU* function, as many times as there are MCUs in the entropy-coded segment.

After the processing of the entropy-coded segment has finished, the next marker is read from the compressed image data. If this marker is a restart marker, a restart is made. This is done by resetting the prediction of the Huffman decoder and the bits stored in the bit ASU for the GetBits procedure. Then the entropy-coded segment is again processed and the next marker is read.

When the found marker is not a restart marker, then it is assumed that all entropy-coded segments are processed, thus also the last entropy-coded segment. In accordance with the data flow in the processing of the SOS marker (see Figure 2-2), we end the processing of the SOS marker.

The only important function not yet explained is the *ProcessMCU* function. This function is responsible for processing a minimal coded unit. It therefore has to process every block in the MCU. This means that it must perform a variable length decoding, zigzag scan and dequantization to it. After that, the block must be sent to the communication channel.
The set up of the *ProcessMCU* function is now as follows. The function executes the *ProcessBlock* function, as many times as there are blocks in the MCU being decoded. The *ProcessBlock* function is responsible for performing al necessary operations on the block, including the sending of it to the communication channel. The *ProcessBlock* will be discussed in detail in the following section.

**VLD / ZZ / DQ**
The variable length decoder, zigzag scan and dequantization unit are implemented in one C function: *ProcessBlock*. The function definition is given by the following C code.

```
void ProcessBlock(const Uint<4> curcomp,
                  const Uint<8> MCU_row,
                  const Uint<8> MCU_column,
                  Uint<12> ComOut[68]);
```

**Figure 8-10 ProcessBlock function**

The function must be supplied with the row and column number of the MCU being decoded. The *curcomp* variable indicates the number of the block inside the MCU. The function of the *ComOut* array has already been explained in the implementation of the output.

When the *processBlock* is executed, it fills in the first four variables of the communication packet, being a block packet. This is done by the executing the following statement.

```
ComOut[0] = 0x800;
ComOut[1] = MCU_column;
ComOut[2] = MCU_row;
ComOut[3] = curcomp;
```

**Figure 8-11 Initialization of communication packet**

The *processBlock* function then extracts the DC element for this block from the compressed image data stream. This DC element is then decoded and reformatted to a 2' s complement notation. The result is then stored in the variable *value*. These operations are performed by the following code.

```
Uint<4> select = com_ram_read(com_ram_pos_MCU_valid + curcomp);
Uint<8> symbol = get_symbol(DC_CLASS, com_ram_read(com_ram _pos_comp_DC_HT
                            + select));
Uint<1> ErrorFlagBitasu;
Uint<16> bits = get_bits(symbol, ErrorFlagBitasu);
ErrorBit |= ErrorFlagBitasu;
Int<11> value = reformat(bits, symbol);
```

**Figure 8-12 Decoding of DC symbol**

The decoded element does no have the value that represents the difference between the original value of the first element of this block and the value of the previous block. As specified, this should be undone. Therefore, the following statement adds the value of the DC element the previous block to the correct value and then stores this new value to the variable *prediction*. This is done by the following two statements:

```
value += com_ram_read(com_ram_pos_comp_PRED + select);
com_ram_write(com_ram_pos_comp_PRED + select, value);
```

**Figure 8-13 Decoding of DC symbol (continued)**

The *processBlock* function is now ready to start decoding the AC elements of the block. This is done in the following way. A global variable *zero_run* is set to zero. Then the number of the AC Huffman table that should be used is retrieved from the shared memory. In addition, it calculates the dequantization value. A for loop is then passed 64 times (for every element in the block one pass). If the *zero_run* variabel equals zero, the variable *outp* is set to value, else it is set to zero. Every time the *zero_run* variable equals zero a new symbol is retrieved and decoded from the compressed image data stream. The *zero_run* variable is then set to the value indicated by the symbol decoded. In addition, the variable *value* is set to the value of the decoded symbol.

At the end of the for-loop the position of the element being decoded is looked up using an array that contains the elements position after the zigzag scan. The output value is then dequantized by multiplying the *outp* value with the correct dequantization value. The result of this operation is stored in the correct location of the *ComOut* array.

When the for-loop has ended, the following statement is executed:

```
com_write_fifo(ComOut);
```

**Figure 8-14 Sending of block packet to channel**

This statement writes the block packet to the communication channel. The block is then completely processed and is send to the communication channel. The *processBlock* function can thus end.

## Bit ASU

A description of the functional behavior of the bit ASU was given in chapter 5.5. This functional behavior of the bit ASU has been implemented into two C models. Both models use the same function headers to call the different procedures present in the bit ASU. The function headers for the different procedures are shown in Figure 8-15.

```
Uint<8> ReadByte(void);

Uint<16> ReadSegmentSize(void);

void ReadMarker(Uint<1>& bit0, Uint<1>& bit1, Uint<1>& bit2, Uint<1>& bit3,
                Uint<1>& bit4, Uint<1>& bit5, Uint<1>& bit6, Uint<1>&
                bit7);

void skip_until_byte_boundary(void);

Uint<16> get_bits(const Uint<8> number, Uint<1>& ErrorBitASU);
```

**Figure 8-15 Function headers for bit ASU functions**

The first model (A|RT model) is a non cycle-true model. The functions in this model provide a functional correct implementation of the procedures supported by the bit ASU. This model produces the correct result after one function call. Therefore, no hold logic has to be implemented into the C model.

The functions used in the A|RT model require access to the compressed image data in order to produce valid results. Whenever one of these functions needs a byte from the compressed image data, it calls the function *get_byte*. This function then returns the first byte from the compressed image data stream that has not been read. The function *get_byte* must be supplied by the bench and has the following header.

```
Uint<8> get_byte(void);
```

**Figure 8-16 get_byte function header**

The second model is a bit more complex. This model (implementation model) is a cycle-true model. It is therefore not guaranteed that the requested procedure can be executed in one cycle (one execution of the function). The implementation model therefore has a hold variable that indicates whether the execution of the procedure has finished.

The implementation model uses the architecture shown in Figure 8-17 to realize a cycle-true C model for the bit ASU. The outputs of the block logic control the multiplexers and selection of the correct procedure. These control lines are not shown in the figure for the sake of readability.

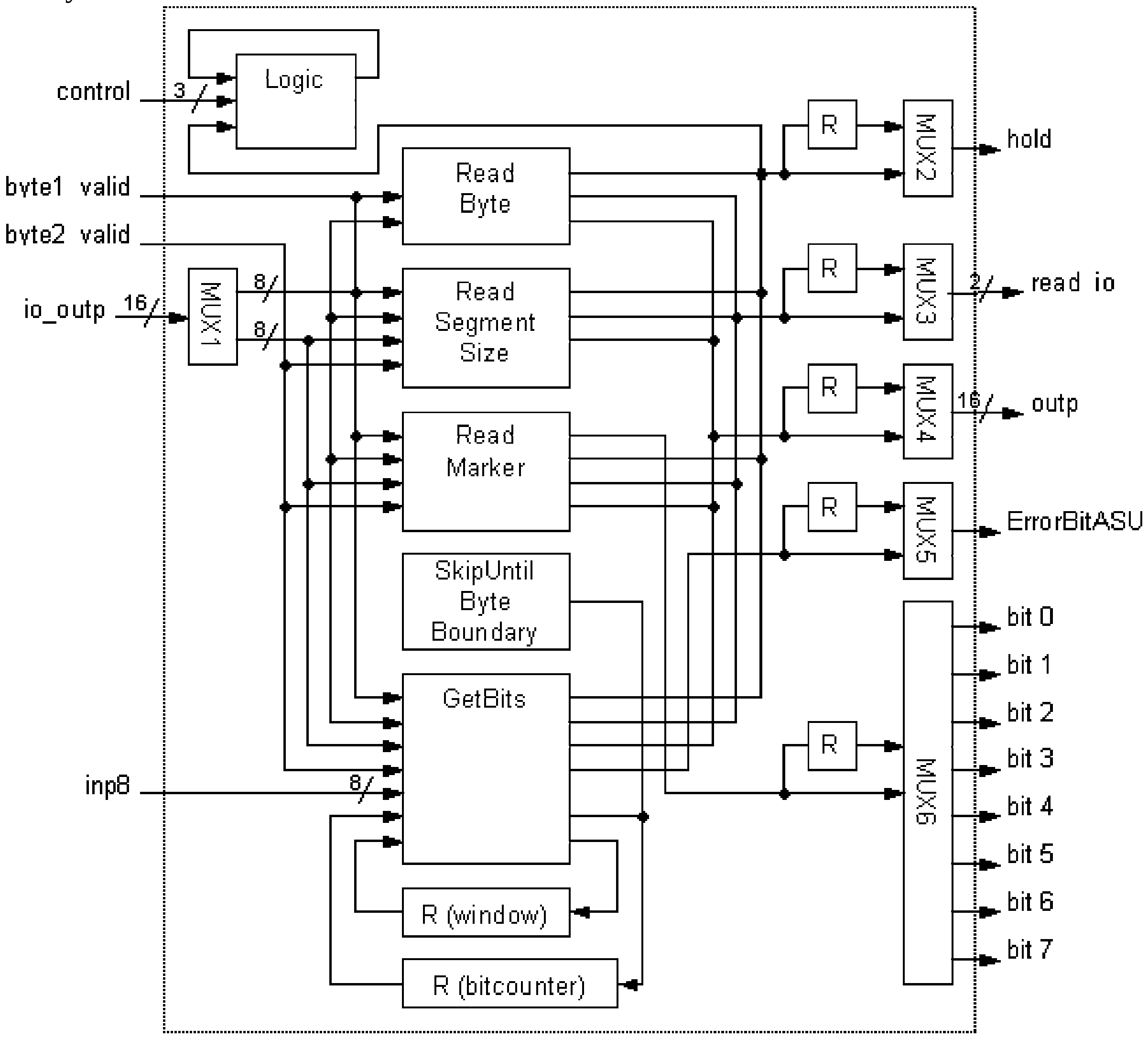


**Figure 8-17 Implementation architecture for bit ASU**

All arrows crossing the dashed box form variables that are part of the interface of the bit ASU. Based on this, the following C header for the bit ASU is constructed:

```
void bitasu(const Uint<3> control,
            Uint<1>& ErrorBitASU,
            const Uint<16> io_outp,
            const Uint<1> byte1_valid,
            const Uint<1> byte2_valid,
            Uint<2>& read_io,
            const Uint<8> inp8,
            Uint<16>& outp16,
            Uint<1>& bit0,
            Uint<1>& bit1,
            Uint<1>& bit2,
            Uint<1>& bit3,
            Uint<1>& bit4,
            Uint<1>& bit5,
            Uint<1>& bit6,
            Uint<1>& bit7,
            Uint<1>& hold);
```

**Figure 8-18 bitasu function header**

The block labeled SkipUntilByteBoundary realizes the corresponding procedure. This procedure can always be executed in one cycle. It sets the variable bitcounter to zero.

The block labeled ReadSegementSize realizes the corresponding procedure. It checks whether the two input bytes are valid. In that case, it computes the segment size; else, it sets the hold value and computes no output.

The block labeled ReadByte realizes the corresponding procedure. It checks whether the first byte of the input is valid. In this is true, it outputs this byte. Else, it sets the hold value and computes no output.

The block labeled GetBits realizes the corresponding procedure. The procedure is shown in Appendix C.

The block labeled ReadMarker realizes the corresponding procedure. The procedure is shown in Appendix D.

All procedures set the *read_io* value corresponding to the number of bytes the have used from the compressed image data to compute the output. Whenever no output has been computed and hold has been set, the *read_io* has a value of zero. Only in the case, that the ReadMarker procedure was executed is it possible that the hold is set and the *read_io* does not have a zero value.

The implementation model has to deal with the possibility that one execution of the model (*bitasu* function) is not enough to calculate the result of the called procedure. Therefore, the model provides us with a *hold* variable. The hold logic, which has to deal with this hold value, must be implemented in the model. A second problem that must be handled by the implementation model is the use of the input block. This input block provides the access to the compressed image data.

Both problems are dealt with in a function called *bitasu_art*. The function body is shown in Figure 8-19. The do/while-loop is executed until the *hold* variable is zero, which means that the *bitasu* function has been executed enough times to calculate the output of the command (bit ASU function) selected by the variable control.
Inside the loop, three function calls are made. The first function call, *IO_art*, retrieves the values for the *io_outp*, *byte1_valid* and *byte2_valid* from the input block. Then the *bitasu* function is called. This function is the bit ASU implementation model. Then another call to the *IO_art* function is made. The *read_io* variable has been set by the preceding *bitasu* function call. The input block can process this value with the *IO_art* function call.

```
do{
   read = 0;
   IO_art(read, io_outp, byte1_valid, byte2_valid);
   bitasu(control,ErrorBitASU,io_outp,byte1_valid,byte2_valid,read,inp8,
          outp16,bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7,hold);
   IO_art(read, io_outp, byte1_valid, byte2_valid);
}while(hold);
```

**Figure 8-19 Implementation of hold logic**

With this solution, calling functions do not have to bother about the hold logic. They know that when the execution of the *bitasu_art* function finishes, the correct results are available. In addition, it keeps the compressed image data hidden for the calling functions.

To allow easy interchangeability of the A|RT model and the implementation model, the functions provided by those models should be the same. Therefore, the implementation model should provide correct implementation of the functions shown in Figure 8-15.

Therefore, all functions shown in Figure 8-15 have been implemented as shown in the example of Figure 8-20.

```
Uint<8> ReadByte(void)
{
   bitasu_art(cmd_readbyte,ErrorBitASU,inp8,outp16,bit0,bit1,bit2,bit3,
              bit4,bit5,bit6,bit7);

   return outp16;
}
```

**Figure 8-20 Example of C function used to implement bit ASU implementation model**

## Input block

The input block has been implemented in C using the architecture shown in Figure 8-21.

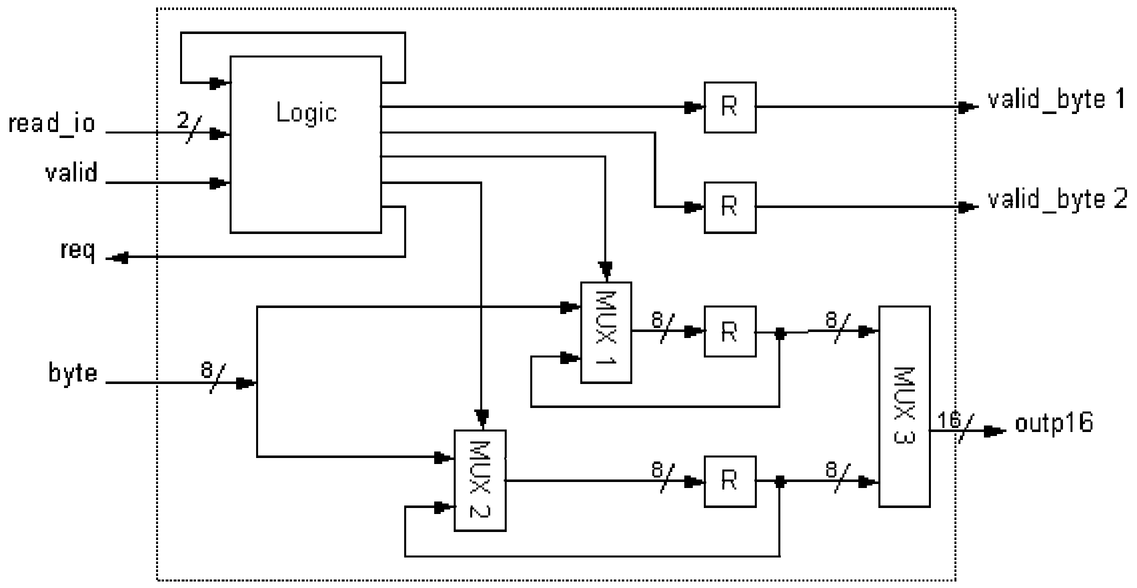


**Figure 8-21 Implementation architecture for input block**

All arrows crossing the dashed box form variables that are part of the interface of the input block. Based on this, the following C header for the input block is constructed:

```
void IO(Uint<1>& req,
        const Uint<8> byte,
        const Uint<1> valid,
        const Uint<1> eof,
        const Uint<2> read,
        Uint<16>& outp16,
        Uint<1>& valid_byte1,
        Uint<1>& valid_byte2);
```

**Figure 8-22 IO function**

The implementation of this function is a direct implementation of the architecture of Figure 8-21. The logic is implemented according to the state transitions functions of Table 5-2.

The bit ASU is the only unit in the bit stream processor that calls the input block function. However, the interface of the IO function contains not only variables that form an interface with the bit ASU. Variables used to communicate with the input of the JPEG decoder (e.g. outside world) are also present in this function. To allow the bit ASU to call the input block with the interface that is visible for it the following function has been added:

```
void IO_art(const Uint<2> read, Uint<16>& outp16, Uint<1>& valid_byte1,
            Uint<1>& valid_byte2)
{
      get_byte(req, byte, valid, eof);
      IO(req,byte,valid,eof,read,outp16,valid_byte1,valid_byte2);
}
```

**Figure 8-23 IO_art function**

The function header matches the interface between the bit ASU and the input block. Inside the function, a call is made to *get_byte*. This function is responsible for providing the first unread byte from the compressed image data when the *req* variable is one. The result of *get_byte* call is that the variable *byte* contains a new byte from the compressed image data stream if requested. This depends on the *req* value from the previous IO function call. The second function called in the *IO_art* function is the IO function that realizes the input block.

### 8.1.3. Image processor

The functional behavior of the image processor has been described in chapter 6. This was done by describing the functional units and their relationship. We will explain the C models used in our project for these units in this section. The C model for the image processor can be found in the files imagep.cxx and imagep.h

#### Memory

In Figure 6-1, a memory is placed between the IDCT and color-conversion unit. This was done to store blocks that have passed the IDCT unit and are waiting to be processed by the color conversion unit. This memory is implemented in the C model using an array named *MCUbuff*. This array must be able to store the maximum amount of blocks that can form one MCU. This is because the color conversion only starts when the complete MCU is available. The specification declares that therefore a storage capacity of ten blocks is needed. For that reason, the size of the *MCUbuff* is set to 640 elements.

#### IDCT

The C model of the IDCT uses a two-dimensional inverse discrete cosine transform that is performed using a repeated one-dimensional inverse discrete cosine transform. For this one-dimensional IDCT an implementation of the Chen algorithm is used. The complete IDCT implementation was developed in another project and documentation about this implementation can be found in [2]. The results of the IDCT unit are written into the *MCUbuff*.

#### Color conversion and reordering

The color conversion and reordering unit are combined into one function. This is called the *imagep_process* function. The function definition is given by:

```
void imagep_process(const Uint<8> MCU_row,
                    const Uint<8> MCU_column,
                    Uint<8> FrameBuffer[MAX_FRAMEBUFFER_SIZE]);
```

**Figure 8-24 imagep_process function**

This function takes as an argument the MCU row and column number that must be processed. With these numbers, the function can locate the position of the MCU being processed in the image.

The reconstructed image is stored in the *FrameBuffer*. This *FrameBuffer* is supplied to the *imagep_process* function. In that way, the function can write all MCU data to the correct location in the image.

**Control logic**

The functional behavior of the control logic of the image processor is implemented in the C function *imagep*.

The *imagep* function reads communication packets from the communication channel using the *com_fifo_read* function. These packets are then stored in the *ComOut* array. The memory space for this array is supplied by the function that called the *imagep* function.

Every time the communication packet contains a block packet, the IDCT function is called. After the function has finished it is checked whether all blocks of the MCU being decoded are present in the *MCUbuff*. If all blocks are indeed present in the *MCUbuff* array, the *imagep_process* function is called.

In case the retrieved communication packed contained an EOI communication packet, the *imagep* function ends with retrieving the width, height, number of color components in the image and the error flag from the shared memory.

### 8.1.4. JPEG decoder

The C implementation of the JPEG decoder is done in the following way. First, the bits stream processor must be executed. This is done by calling the function *bitsp*. When this function has finished, the image processor must be started. This is then done by calling the *imagep* function.

## *8.2. Test bench*

The previous paragraph described the C model developed for the bit stream processor, image processor and the JPEG decoder. To prove that the models are correct, a set of test benches has been made. This set contains a bench for the bit stream processor (bitsp_bench.cxx), a bench for the image processor (imagep_bench.cxx) and a bench for the JPEG decoder (jpeg_bench.cxx). These test benches can be found in the deliverable under the directory src/jpeg_pc2. The directory src/tests of the deliverable contains input and reference files for all test benches.

The bit stream processor bench requires a valid JPEG picture in ASCII format. One set of input/reference files is available in this format. The file philips.INP should be used as input and the files ComRam.REF and ComOut.REF must be used as reference.

The files ComRam.INP and ComOut.INP should be used as an input for the image processor bench. The file jpeg_ref.ras must then be used as a reference file for the result of the bench.

The JPEG decoder bench requires a valid JPEG picture in ASCII format. Three sets of input/reference files are available in this format. The first set consists of philips.INP and jpeg_ref.ras. The second set contains bj.INP and bj_ref.ras. The third set contains et.INP and et_ref.RAS.

A number of important image characteristics of these input files (JPEG images) are shown in Table 8-1.

**Table 8-1 Properties of test images**

| Image | Size [pixels] | #Colors | Color format | File size [bytes] |
|---|---|---|---|---|
| philips.INP | 50x67 | 3 | 4:1:1 | 15000 |
| et.INP | 600x398 | 3 | 4:1:1 | 91772 |
| bj.INP | 1600x1200 | 3 | 2:1:1 | 296952 |

Both the bit stream processor bench and JPEG decoder bench can be compiled using the A|RT model (bitasu.cxx) or the implementation model (bitasu_impl.cxx) for the bit ASU. In that way, both models can be simulated.

# 9. VHDL model

The Very High Speed Integrated Circuit Hardware Description Language [VHDL] is one of the standard languages used to describe a system at a cycle-accurate level. Using the A|RT tools a VHDL model has been made for the bit stream, image processor and JPEG decoder. The VHDL model for both processors and the JPEG decoder are described in the next paragraph. A set of test benches is presented in paragraph 9.2. Using these test benches, simulations of all models have been made. Experimental results obtained from these simulations are presented in paragraph 9.3.

## 9.1. Implementation

This paragraph describes the VHDL implementation of the bit stream, image processor, communication logic and JPEG decoder.

### 9.1.1. Communication logic

The communication logic consists of two units, the communication channel and the communication RAM arbiter. A C model was presented for both units in chapter 8. The single cycle C functions *com_buffer* and *com_ram_rw* implemented the units.

These C models need to be converted to VHDL. This can be done with A|RT Builder, which is capable of translating single cycle C functions into VHDL.

With A|RT Builder, VHDL models for the communication RAM arbiter (*com_ram_arbiter*) and the communication channel (*com_buffer*) have been generated.

In the C model of the communication channel is it possible to select the number of communication packets that can be stored in the channel. For the VHDL model, we set this at a value of two.

These VHDL models can be found in the file com.vhd in the directory src/jpeg_pc2/vhdl of the deliverable. The interfaces of the entities are shown in Figure 9-1 and Figure 9-2.

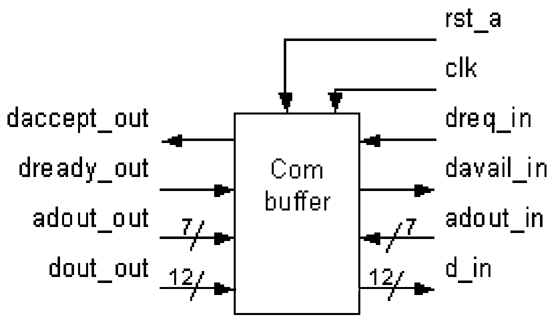


**Figure 9-1 Interface of com_buffer component**

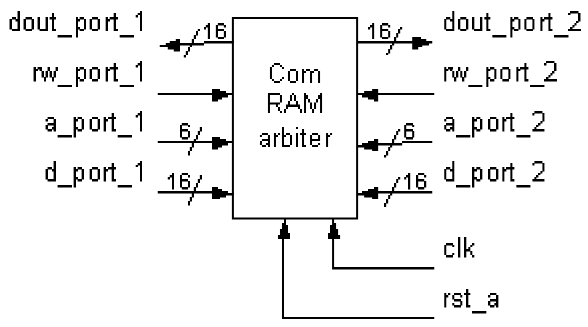


**Figure 9-2 Interface of com_ram_arbiter component**

### 9.1.2. Bit ASU

A VHDL model for the bit ASU can be generated using A|RT Builder. The *bitasu* function from the bit ASU implementation model (see page 35) is a single cycle C function that implements the bit ASU. It can therefore be translated with A|RT Builder in a VHDL component *bitasu* that implements the bit ASU.

The VHDL model for the *bitasu* component can be found in the files bitasu_a.vhd and biasu_e.vhd in the directory src/jpeg_pc2/artd/jpeg_lic/bitasu of the deliverable. The interface of the component is shown in Figure 9-3.

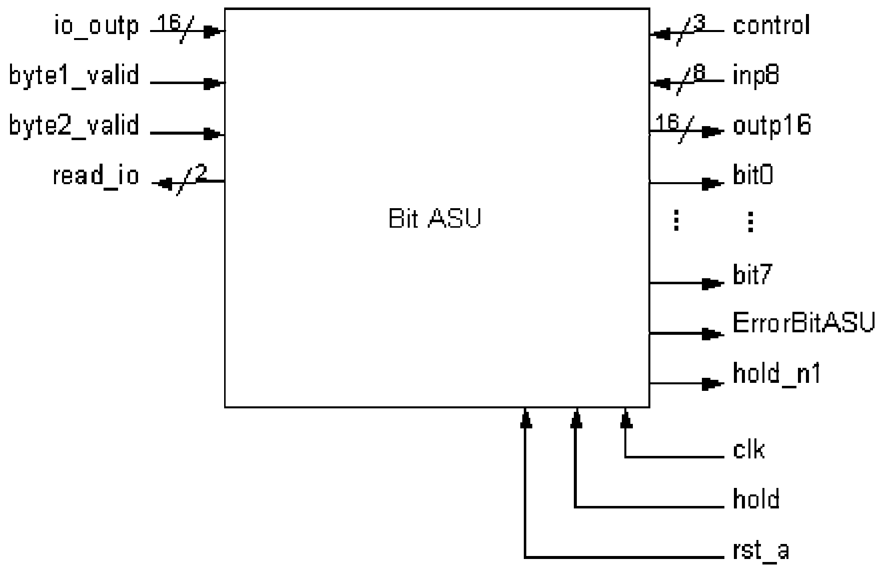


**Figure 9-3 Interface of bitasu component**

### 9.1.3. Input block

A VHDL model for the input block can be generated using A|RT Builder. The *IO* function from the input block model (see page 35) is a single cycle C function that implements the input block. It can therefore be translated into a VHDL component *IO* that implements the input block using A|RT Builder.

The VHDL model for the IO component can be found in the file inputblock.vhd in the directory src/jpeg_pc2/vhdl of the deliverable. The interface of the component is shown in Figure 9-4.

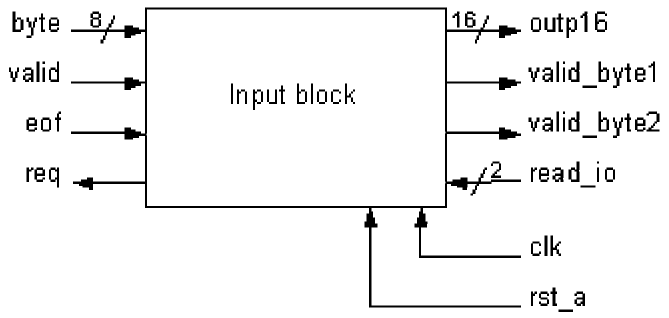


**Figure 9-4 Interface of IO component**

## 9.1.4. Bit stream processor

The C model for the bit stream processor is described in chapter 5. This C model must be converted into a VHDL model. This can be done using A|RT Designer Pro. A|RT takes the C model and an architecture as its input. It then maps the algorithm onto the architecture and performs a scheduling on the operations. A|RT can then write the design out in the form of a VHDL component. This component is then a processor that uses the described architecture and runs the algorithm (C model).

We take the C model of the bit stream processor as an input to A|RT. The architecture used in A|RT for the bit stream processor is shown in Figure 9-5.

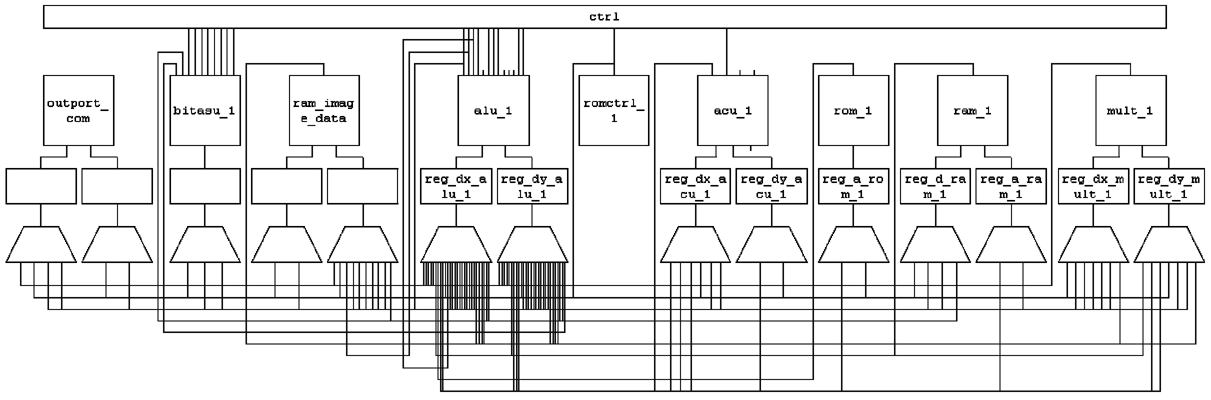


**Figure 9-5 Architecture of bit stream processor**

All blocks, except the *bitasu_1* and *ram_image_data*, are standard functional blocks. The *bitasu_1* block implements the bit ASU. The *ram_image_data* block implements the communication RAM arbiter. These two blocks can be found in the JPEG library (jpeg_lib). This library can be found in the directory src/jpeg_pc/artd of the deliverable.

A number of outputs of the bitasu_1 block go directly to the controller. These outputs are the *bit0* through *bit7* outputs. A|RT detects that these outputs must be used by the controller as flags. Upon detection of a marker, the bitasu_1 sends its output to the controller. The jump logic of the controller can then directly decide to which program counter is must jump.
The gain of this solution is that it is not necessary to perform compare operations on the ALU to decide which marker was detected and to which instruction the controller must jump. The next program counter can now be determined in one cycle, which is the fastest solution possible.

With this architecture, C model and A|RT Designer a VHDL component called *artd_bitsp* is generated. This component implements the bit stream processor in VHDL. All files needed to run A|RT Designer and generate a VHDL component for the bit stream processor can be found in the directory src/jpeg_pc2/artd/bitsp of the deliverable. The C model needed by A|RT can be found in the src/jpeg_pc2 directory.

There are however two problems with this result. The first problem is that the *ram_image_data* block is kept inside the processor. This block should however be external; we can then connect both the bit stream and image processor to it. This block can be made external by using a make_external pragma for it in the HDL generation step.

The second problem is a bit more complex. This involves the *bitasu_1*. The block implements the bit ASU. From the point of view of the bit stream processor, provides this block access to the compressed image data stream. The bit ASU must therefore have access to this compressed image data stream. In the functional description of the bit stream processor, was defined that this access should be implemented via an input block (see paragraph 8.1.2). This input block has two interfaces. One is used to connect to the bit ASU; the other interface is connected to the outside world.
This input block and the connection of it to the bit ASU and outside world are not made by A|RT. This block and the connections are however necessary for the bit stream processor to function correctly and can therefore not be omitted. Subsequently, the VHDL that is generated by A|RT should be adapted, such that the necessary changes are made to the *artd_bitsp* component.

The list of changes that must be made to the *artd_bitsp* component can be found in Appendix E. These changes can be made to the A|RT VHDL output by running the vhdl_adap script. This script can be made though the compilation of the file vhdl_adap.cxx. This file can be found in the directory src/jpeg_pc2/vhdl of the deliverable.

The interface of the *artd_bitsp* component (bits stream processor) is shown in Figure 9-6.

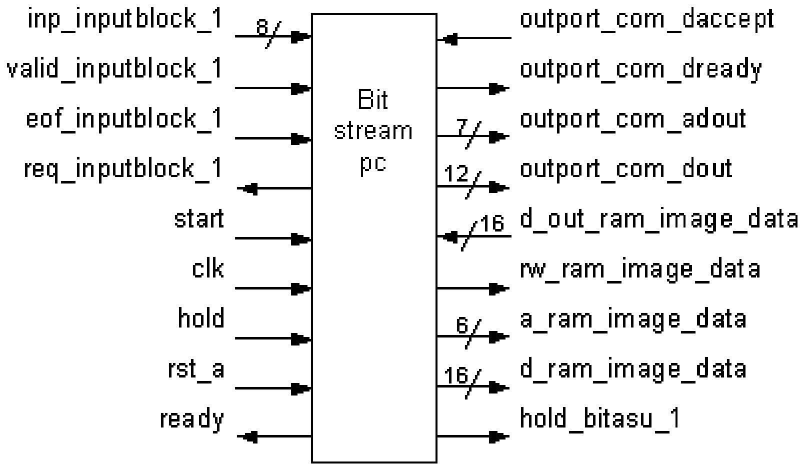


**Figure 9-6 Interface of artd_bitsp component**

### 9.1.5. Image processor

The C model for the image processor is described in chapter 6. This C model must be converted into a VHDL model. This can be done in the same way as with the bit stream processor.

We take the C model of the image processor as an input to A|RT. The architecture used in A|RT for the image processor is shown in Figure 9-7.

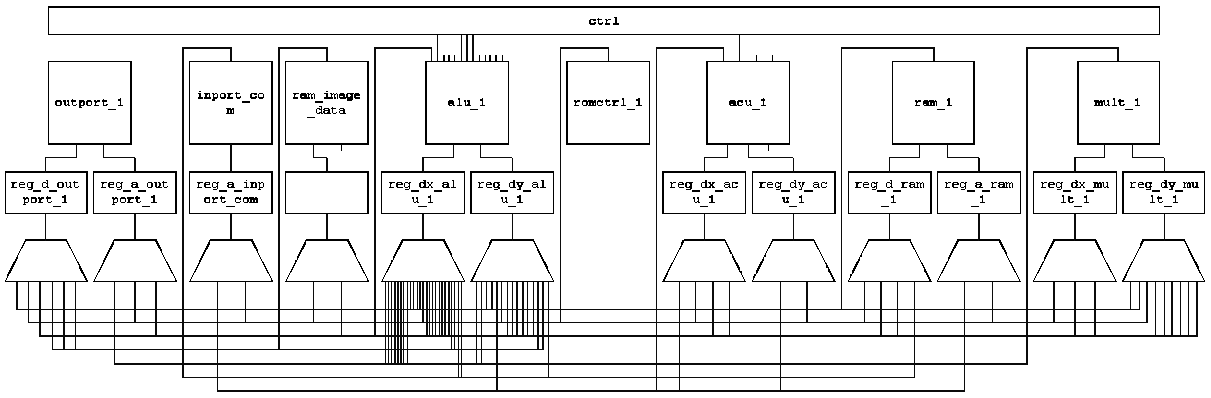


**Figure 9-7 Architecture of image processor**

All blocks, except the *ram_image_data*, are standard functional blocks. The *ram_image_data* block implements the communication RAM arbiter. This block can be found in the JPEG library.

Just as with the bit stream processor, we need to have the *ram_image_data* block external. This can be done using a make_external pragma for it in the HDL generation step.

A VHDL component called *artd_imagep* is generated using A|RT Designer. This component implements the image processor in VHDL. All files needed to run A|RT Designer and generate a VHDL component for the image processor can be found in the directory src/jpeg_pc2/artd/imagep of the deliverable. The C model needed by A|RT can be found in the src/jpeg_pc2 directory.

The interface of the *artd_imagep* component (image processor) is shown in Figure 9-8.

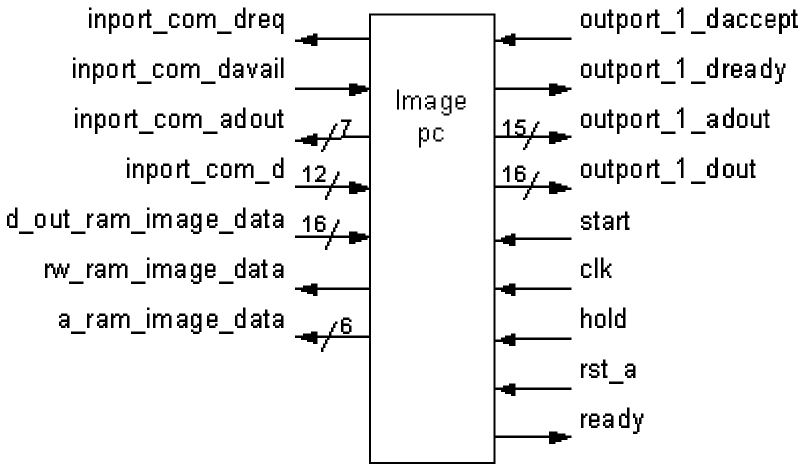


**Figure 9-8 Interface of artd_imagep component**

## 9.1.6. JPEG decoder

The general architecture of the JPEG decoder is shown in Figure 3-3. A component called *artd_jpeg* implements this architecture into VHDL. The *artd_bitsp*, *artd_imagep*, *com_buffer* and *com_ram_arbiter* components are grouped into this *artd_jpeg* component. The connections between the various components are shown in Figure 9-9. The arrows crossing the dashed box form the ports of the *artd_jpeg* component.

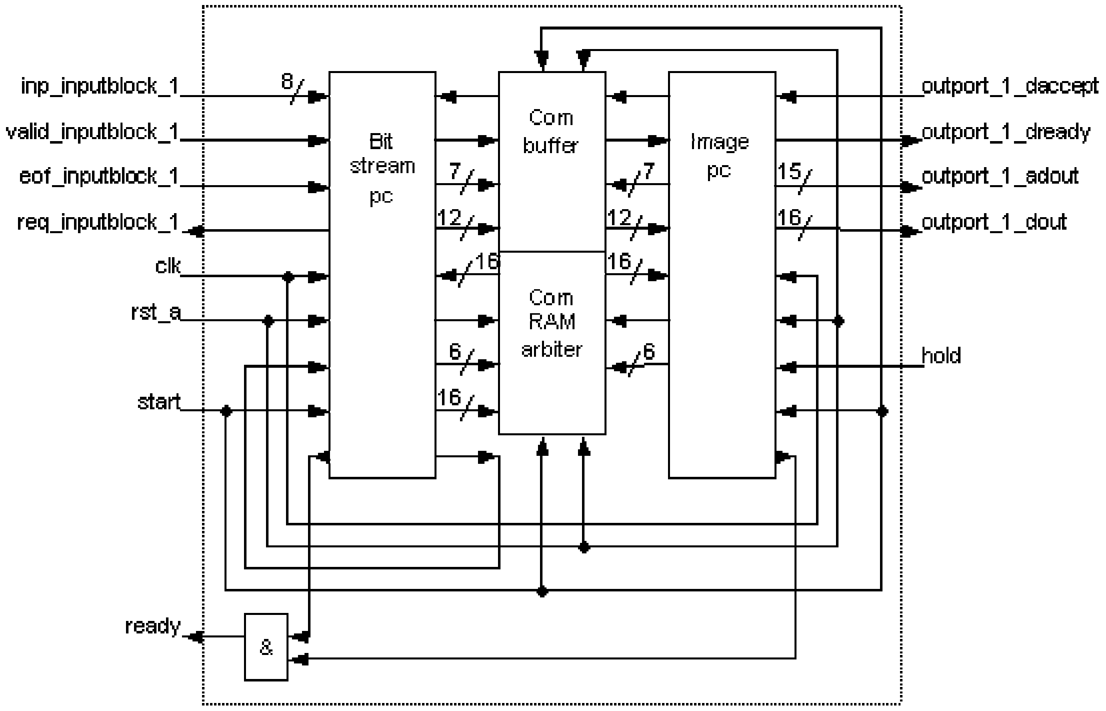


**Figure 9-9 Architecture of artd_jpeg component**

The VHDL model for the *artd_jpeg* component can be found in the file jpeg.vhd in the directory src/jpeg_pc2/vhdl of the deliverable.

## 9.2. Test bench

The previous paragraph described the VHDL model for the bit stream, image processor and JPEG decoder. To prove that the models are correct, a set of test benches has been written. This set contains a bench for the bit stream processor (artd_bitsp_bench.vhd / artd_bitsp_bench.cfg), a bench for the image processor (artd_imagep_bench.vhd / artd_imagep_bench.cfg) and a bench for the JPEG decoder (artd_jpeg_bench.vhd / artd_jpeg_bench.cfg). These test benches can be found in the deliverable under the directory src/jpeg_pc2/vhdl.

The input and reference files for the bit stream processor test bench are the same as with the C model test bench for this processor (see page 44).

The files ComRam.INP and ComOut.INP should be used as an input for the image processor bench. The file FrameBuffer.REF must be used as a reference file for this bench.

The VHDL test bench for the JPEG decoder can only take the philips.INP file as an input. The FrameBuffer.REF file should be used to verify the results of the bench.

## 9.3. Experimental results

Using the test benches described in the previous paragraph, simulations of the bit stream, image processor and JPEG decoder have been made. These simulations were made using ModelSim 5.4e. The experimental results obtained from these simulations are presented in this paragraph.

The most important result from the tests is that the VHDL models for the bit stream, image processor and JPEG decoder produce outputs that not differ from the reference files. From this we may conclude that the VHDL models are correct.

With the simulations, we obtained also some performance characteristics on the designs. These characteristics are presented in Table 9-1. The first column contains the design that was simulated. The second column lists the number of cycles needed to perform the actions of design under test on the compressed image data. The third column lists the number of cycles needed to initialize the design under test. The last column lists the time needed to run the complete simulation (decompression of one image).

**Table 9-1 Performance characteristics**

| Design | #Cycles/image | #Cycles init | Time/image [min:sec] |
|---|---|---|---|
| Bit stream | 368489 | 6 | 9:43 |
| Image | 528463 | 2 | 8:10 |
| JPEG decoder | 676562 | 6 | 27:44 |

# 10. SystemC

SystemC is a C++ class library and a methodology that can be used to create a cycle-accurate model of software algorithms, hardware architecture and interfaces of a SoC (System On a Chip).

The possibility to simulate at a cycle-accurate level using SystemC provides the designer with the possibility to simulate a system at the same level as with VHDL. SystemC however uses an executable to simulate the system, where VHDL uses a VHDL simulator. The use of this executable allows very fast simulation of the design at a cycle-accurate level.

A|RT Designer can generate a SystemC model from a C code input description. The concept of the SystemC model outputted by A|RT is discussed in the next paragraph.

SystemC models have been made for the bit stream, image processor and JPEG decoder. These models are described in paragraph 10.2. To verify the behavior of the models, test benches have been made for the bit stream, image processor and JPEG decoder. These test benches are presented in paragraph 10.3. Experimental results obtained from simulations with these test benches are presented in paragraph 10.4.

## *10.1. SystemC in A|RT*

A|RT Designer can generate a SystemC model from a C code input description. The concept of the SystemC model outputted by A|RT is discussed in this paragraph. Basic SystemC concepts are used during this discussion. For an introduction in these basic SystemC concepts, one should read the first chapters of the SystemC manual [3].

The SystemC model outputted by A|RT Designer is a set of C++ files. These files can be found in the artd_cxx directory inside the A|RT project directory. These files contain a C++ class that describes the cycle-accurate behavior of the A|RT Designer processor. This A|RT Designer processor is grouped inside a SystemC module. The SystemC module itself is placed inside a SystemC bench that can be used to verify the behavior of the A|RT Designer processor. This hierarchy is shown in Figure 10-1. The C++/SystemC code used for the implementation of all levels is discussed in the following sections.

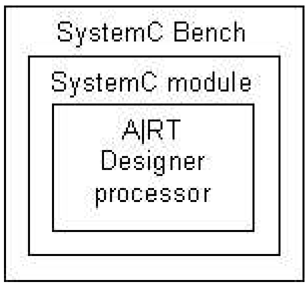


**Figure 10-1 Overview of A|RT SystemC output**

Let us assume that we have generated a SystemC model for a processor *pc1*. The files artd_pc1.cxx and artd_pc1.h will then contain a C++ class *artd_pc1*. This class is a C++ model of the A|RT Designer processor. It contains an instance of all functional blocks used in the processor. Every instance of a block is a C++ class that describes the functional behavior of this block.

A processor in general uses inputs and outputs to communicate with the outside world. These inputs and outputs are also functional blocks (e.g. inport and outport) that are present inside the processor. These blocks have however not only connections inside the processor. They also have connections to the outside world. To create these connection, the constructor of the C++ class that describes the A|RT processor must be supplied with a references to instantiations of all functional block inside the processor that have inputs and/or outputs (e.g. communicate with the outside world). These references are then linked to a pointer that points to an instantiation of the functional block inside the C++ class. The pointer can then be used inside the C++ class that describes the A|RT processor to access the functional block.

The above-described concept is shown in the following example.

```
class artd_pc1 {
  artd_pc1(
    Ctc_inport<2,1,1> *inport_1 = an_inport_1,
    Ctc_outport<3,1,1> *outport_1 = an_outport_1
  );

  Ctc_inport<2,1,1> *inport_1 = an_inport_1;
  Ctc_outport<3,1,1> *outport_1 = an_outport_1;

  ....
};
```

**Figure 10-2 Example of input and output binding used in A|RT SystemC**

In this example, the processor *pc1* uses one input (*inport_1*) and output (*outport_1*). A reference to this input and output is supplied using the variables *an_inport_1* and *an_outport_1*. These references are linked to variables *inport_1* and *outport_1*. The class *artd_pc1* can then use the variables *inport_1* and *outport_1* to access the input and output supplied by the constructor.

The C++ class that describes the A|RT processor further contains a function *run*. When *run* is called, it executes the processor model for one cycle The exact timing model used inside the *run* function will be discussed at the end of this paragraph.

A|RT further generates the files artd_pc1_ctrl.cxx and artd_pc1_ctrl.h. The files contain a class *artd_pc1_ctrl*. This is a C model of the controller needed for the processor *pc1*. An instance of this class is used in the processor model (class *artd_pc1*), as it is one of the functional blocks inside the processor.

A|RT has generated with these files a C++ class that describes the processor. This class description can be used to simulate just the processor. A bench is therefore generated by A|RT. This bench can be found in the file artd_bench.cxx.

The A|RT SystemC output discussed until now deals with the lowest hierarchical level of Figure 10-1, the A|RT Designer processor. The C++ classes at this level use SystemC data types, but the most important concept of SystemC is not used. That is the concept of modules. Modules are the basic building blocks within SystemC to partition a design. Modules allow designers to hide internal data representation and algorithms from other modules. This forces the designer to use public interfaces to other modules and that makes the entire system easier to change and easier to maintain. At the next hierarchical level of Figure 10-1 and the A|RT SystemC output, we see such a module for the A|RT Designer processor.

Let us revert to our example processor *pc1*. A|RT generates a SystemC module representation for this processor. The SystemC module is called *artd_pc1_module*. The declaration of the module contains a list of ports that are made public by the module. These ports form the interface of the module. The interface of the *artd_pc1_module* is shown in Figure 10-3.

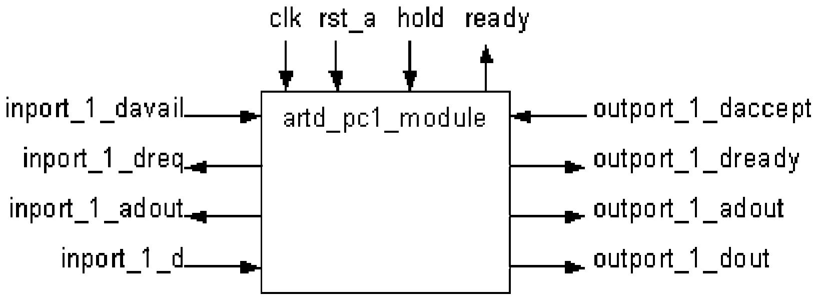


**Figure 10-3 SystemC module artd_pc1_module**

This module has to realize the functional behavior of the A|Rt designer processor. It therefore contains an instance *d_processor* of the class *artd_pc1*. To instantiate the class *artd_pc1*, a reference to the used input and output must be supplied. The SystemC module contains for that reason a C++ classes for every functional blocks that is inside the A|RT processor and has inputs and/or outputs to the outside world (e.g. inport and outport). These C++ classes implement the cycle-accurate behavior of these functional blocks. They further provide a conversion between the data types used in the A|RT processor and the data types used in the ports of the module. These C++ classes are called wrapper classes in the A|RT SystemC implementation. The name of these classes therefore ends with *_wrap*.

A function *proc_clk* is also present in the module. This function calls the *run* function of the A|RT processor when the signal supplied to the module via the *clk* port is one. The A|RT processor is then run for one cycle.

To construct the module a constructor must be supplied. The SystemC keyword SC_CTOR must be used to do this. The constructor for the module *artd_pc1_module* is given in the following example.

```
SC_CTOR(artd_pc1_module):
  d_inport_1_wrap("inport_1",this),
  d_outport_1_wrap("outport_1",this),
  d_processor(&d_inport_1_wrap, &d_outport_1_wrap
  )
{
  SC_METHOD(proc_clk);
  sensitive << clk;
}
```

**Figure 10-4 SystemC constructor for module artd_pc1_module**

The constructor defines one process using the SystemC keyword SC_METHOD. This specifies that the module contains a process named *proc_clk*, this process is triggered by events. By adding the port named *clk* to the sensitivity list, the module is made sensitive for changes of the value of this port. The *proc_clk* function is now called whenever the signal at this port changes.

The SystemC module for the processor *pc1* can be found in the file artd_pc1_module.h. The functions declared by the module are defined in the file artd_pc1_module.cxx.

With this, we end our discussion of the second hierarchical level in the A|RT SystemC output (see Figure 10-1). The highest hierarchical level of Figure 10-1 is formed by a SystemC bench. This bench is present in the A|RT SystemC output to simulate the SystemC module that contains the A|RT processor. This bench can be found in the file module_bench.cxx. It contains SystemC modules for all blocks that must be connected to the processor (e.g. input and output), clock and controller that to control all SystemC modules in the bench.

The test bench present in the A|RT SystemC output for the processor *pc1* is shown in Figure 10-5.

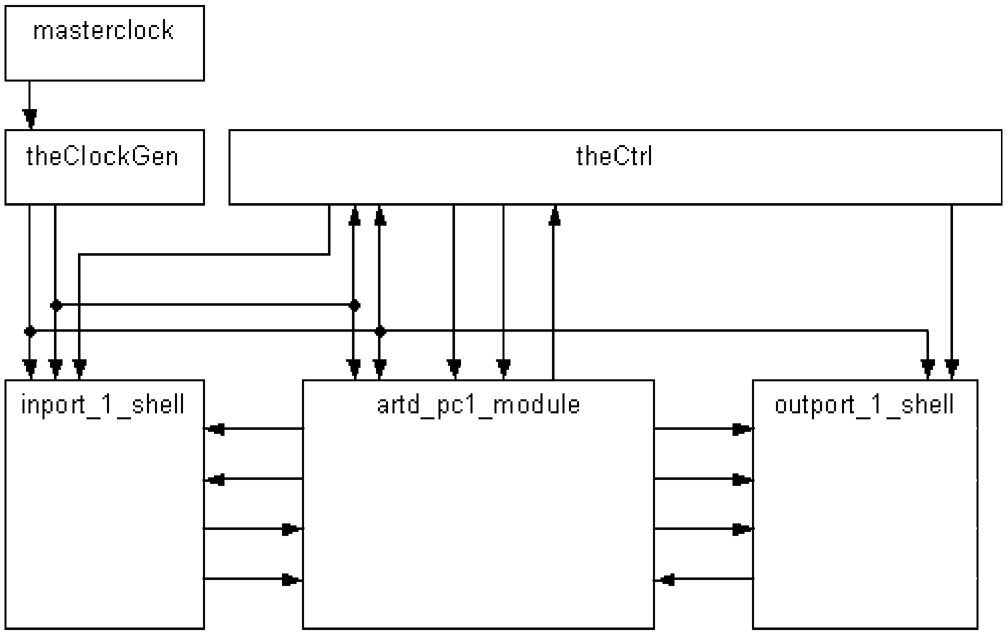


**Figure 10-5 SystemC test bench for SystemC module artd_pc1_module**

This test bench is realized in SystemC by grouping all SystemC modules in the system into a net list. This net list must further contain a list of signals that are used to connect the modules to each other. The connections between the ports of the modules are also declared in the net list. The complete net list is contained in the function *sc_main*.
Running the simulation of the system is then done by calling the *sc_start* function with the number of cycles that should be simulated. For an infinite number of cycles, one should supply a negative value.

We have now described the general set up of the SystemC implementation of A|RT. Let us now take a closer look at timing model used in the *run* function that is declared in the C++ class that models the A|RT processor.

This function has to execute all functional blocks in the A|RT processor for one cycle. It therefore has to execute the functions that model the behavior of the functional blocks which are inside the A|RT processor and are scheduled to be executed at the current program counter. This program counter is stored in a variable *d_PC* and is updated by the C++ class that implements the controller used in the A|RT processor.

One cycle consists of a high clock signal, a transition to a low clock signal, a low clock signal and a transition to a high clock signal again. The functional block must of course be executed when the clock is high and state transitions should be performed when the clock moves from low to high. When the clock is low or moves from high to low, no action is required. It might therefore seem logical to first perform the state transitions and then execute the functional blocks inside the *run* function. A|RT however uses a slightly different method to model the timing inside the processor. The A|RT timing model used in the run function is shown in Figure 10-6.

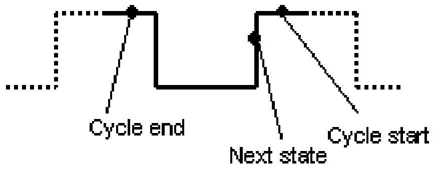


**Figure 10-6 Timing diagram used in A|RT SystemC**

When the function *run* is executed, it first calls the *cycle_end* function of every functional block scheduled on the current program counter. After that, it is assumed that the clock goes to low and again from low to high. If processor is then not on hold, it performs the state transition using the next_state function of all functional units scheduled at this program counter. This includes the functional block of the controller in the A|RT processor, its *next_state* function updates the program counter. Then the *cycle_start* function of every functional block scheduled on this new program counter is executed.

It might seem as both the *cycle_end* and *cycle_start* function are meant to do the same thing and that therefore one can be omitted. This is however not true for the case that we are dealing with functional block that have connections to the outside world (input / output). The functional block *inport* for instance. This block is used to request inputs from the outside world. It must supply the A|RT processor during a cycle with data requested by the A|RT processor during that current cycle. This data is however not present inside the processor module, it has to come from another module. The function inside the module that has to supply the data can however only be run after the *run* function of the A|RT processor has finished. This is due to the sequential property of the C language.

If we use a merged version of the *cycle_start* and *cycle_end* functions, we cannot provide the A|RT processor with the requested data during the current cycle. This is because we cannot access the outside world during that cycle. The separation of the high clock signal in the two parts, cycle start and cycle ends, gives us the possibility to solve this problem.
We have to look at Figure 10-7 to see why this works.

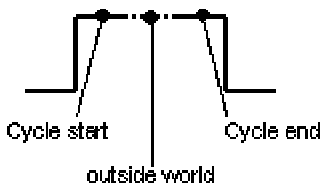


**Figure 10-7 Timing diagram used in A|RT SystemC**

This figure shows the timing diagram of Figure 10-6 from the perspective of the SystemC modules. From that point of view, is the A|RT processor executed using the run function. This ends with the execution of the *cycle_start* function of all scheduled blocks inside the processor. During this *cycle_start* function execution, all blocks inside the processor can send outputs to the outside world.
Then the function *run* ends and the outputs can be processed by the other modules in the SystemC (e.g. inport and outport in the bench). If the A|RT processor requested input data, these modules can set that data on the correct ports of the A|RT processor. When the A|RT processor is then run for another cycle, it starts with executing the *cycle_end* function for all

scheduled blocks. The block that needed inputs can now reed the data from the input ports in the module and they can use this data to compute their next state. This next state includes the outputs of the block for the interface inside the A|RT processor.

The information calculated by the different functions of the functional blocks inside the A|RT processor are summarized in Table 10-1.

**Table 10-1 Results from functions in functional blocks**

| | |
|---|---|
| Cycle start | output = f (state) |
| Cycle end | next_state = f (state, input) |
| Next state | if (!hold) state = next_state |

With the word *output* in Table 10-1, we mean the data that is produced in the functional block and must be processed by the outside world and not inside the processor. The word *input* is used for data that is used by a functional block and this data is not supplied by the processor, but through the outside world. The word *state* is used for all data supplied by the processor to the functional block. This is a control signal or data from registries or any other information contained inside the processor that is needed by the functional block. The word *next_state* means data that is produced by the functional block and is used as state information by the processor.

The described timing model is implemented in the *run* function using the C code shown in Figure 10-8.

```
run (...) {
   switch (d_PC) {
      // Execute cycle_end function of blocks scheduled on this PC
   }
   if( !hold) {
      switch (d_PC) {
         // Execute next_state function of blocks scheduled on this PC
      }
      switch (d_PC) {
         // Execute cycle_start function of blocks scheduled on this PC
      }
   }
}
```

**Figure 10-8 Function run in the C++ class that models the A|RT processor**

Their is one shortcut defined in the timing model used by A|RT SystemC. This shortcut states that if a functional block needs no inputs, then the *next_state* information may already be outputted in the *cycle_start* function. The *cycle_end* function can then remain empty.

## *10.2. Implementation*

This paragraph describes the SystemC implementation of the bit ASU, input block, bit stream processor, image processor, communication logic and JPEG decoder.

### 10.2.1. Bit ASU

A SystemC implementation of the bit ASU is provided with the C++ class *Ctc_bitasu*. This class contains a *cycle_end*, *cycle_start* and *next_state* function as is described in the previous paragraph.

We have a cycle-accurate C model for the bit ASU. This model can now be used to create correct implementations for the *cycle_start* and *cycle_end* functions of the class *Ctc_bitasu*. We therefore have to divide the C model over those two functions. To do so, we look at the timing model presented in the previous paragraph. The bit ASU uses no inputs and produces no outputs. It only requires state information from the A|RT processor. This previous paragraph ended with stating that in that case, the complete implementation of the block can be written in the *cycle_start* function.

However, it is important to note that the *cycle_start* function is only called when the processor in not on hold. It is however possible that the bit ASU cannot calculate the next state in one cycle. It then sets the processor on hold. The bit ASU itself has however no *hold* input and should therefore be executed every cycle until it has computed the next state. The *cycle_start* function however is not executed when the processor is on hold. The only function executed in that case is the *cycle_end* function. The implementation of the bit ASU must therefore be written inside the *cycle_end* function. The *cycle_start* function can remain empty.

The *next_state* function is used to reset the internal state of the bit ASU.

The complete SystemC model can be found in the file ctc_bitasu.h in the directory src/jpeg_pc2/artd/jpeg_lib/bitasu of the deliverable.

### 10.2.2. Input block

The input block is a functional block that is embedded in the JPEG decoder. The interface of this block is shown in the following figure and its functional behavior was defined in chapter 5.6.

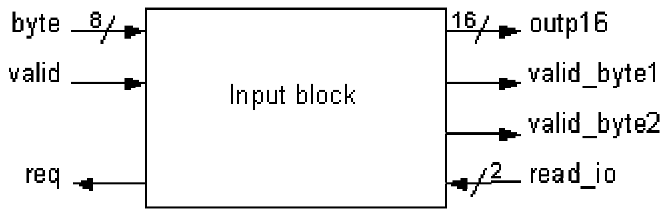


**Figure 10-9 Interface of input block**

A SystemC model for the input block is provided with the class *Ctc_inputblock*. This class contains a *cycle_end*, *cycle_start* and *next_state* function as is described in the previous paragraph.

Before we can create a correct cycle-accurate model for the input block, a problem must be solved. The input block is a functional block with ports as any other block. However, a part of the ports (left hand side of Figure 10-9) is connected to the outside world. In the previous paragraph was discussed that a block that had connections to the outside world should be implemented via a reference to an instance of the block, which is outside the class that describes the A|RT processor. If we would do this for the input block, then the whole interface (left and right hand side of Figure 10-9) will be visible from the outside of the C++ class that models the A|RT processor. However, we want to keep the right hand side invisible from the outside of the C++ class. So, on the one hand we need a part of the interface of the input block to be visible from the outside of the C++ class. On the hand, we want to hide a part of the interface of input block.

To overcome this problem, we introduce the following solution. We define a new block called *inputblock_out*. This block connects to ports of the left-hand side of Figure 10-9. This is shown in Figure 10-10. The signals coming from the input block are copied to the output of the *inputblock_out* block. The signals that form an input for the input block are copied from the input of the *inputblock_out* block to the input of the input block. The *inputblock_out* block is therefore no more than a standard input block. In the previous paragraph was discussed how such a block must be used inside the A|RT processor.

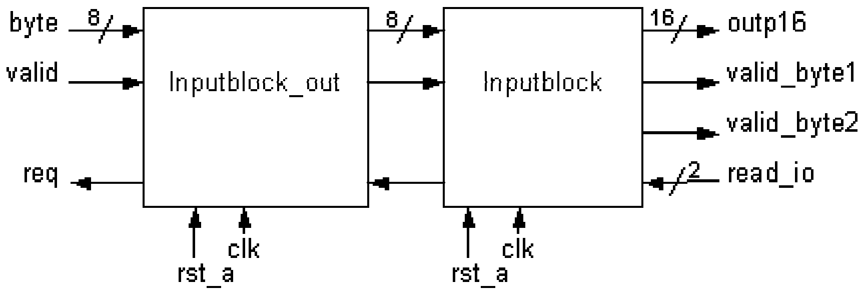


**Figure 10-10 Inputblock_out connected to inputblock**

We can now write valid implementations for the *cycle_end*, *cycle_start* and *next_state* function of the class *Ctc_bitasu*. We have a cycle-accurate C model for the input block. This model can be used to create correct implementations for functions. We therefore have to divide the C model over those two functions. Such a division is however not necessary, as the block has no connections to the outside world. The complete implementation of the behaviour of the block can thus be grouped in the *cycle_start* function.

This *cycle_start* function realizes the functional behaviour of the input block. The C model developed for the input block was already a cycle-accurate model and it can therefore be used in this function. The only change made to the model is that the A|RT data types have been changed into SystemC data types.

The *cycle_end* function is left empty because the calculation of the next state is already performed in the *cycle_start* function.

The *next_state* function is used to reset the internal state of the input block.

We introduced the block *inputblock_out* in this paragprah. This requires of course that we provide a SystemC model for it. The definition of the class *Ctc_inputblock_out* that implements this block is shown in Figure 10-11.

```
class Ctc_input block_out {
   public:
      Ctc_input block_out(const char* name);
      virtual ~Ctc_input block_out();
      virtual void cycle_start(
         sc_ufixed<1,1> req,
         sc_ufixed<8,8> &byte,
         sc_ufixed<1,1> &valid,
             sc_ufixed<1,1> &eof) = 0;
      virtual void cycle_end(
         sc_ufixed<1,1> req,
         sc_ufixed<8,8> &byte,
         sc_ufixed<1,1> &valid,
             sc_ufixed<1,1> &eof) = 0;
      virtual void next_state(bool rst_a) = 0;
   private:
      char* d_name_p;
};
```

**Figure 10-11 class Ctc_inputblock_out**

This class *Ctc_inputblock_out* can now be used in the same way as an input is used. An implementation should be provided in the SystemC module and a reference to an instance of this implementation must be passed to the constructor of the class that models the A|RT processor.

The complete SystemC model can be found in the file ctc_inputblock.h in the directory src/jpeg_pc2/systemc of the deliverable.

### 10.2.3. Communication channel

The communication logic consists of two blocks: communication channel and communication RAM arbiter. We will describe the SystemC implementation of both blocks one after another.

The communication channel has been implemented in the following module:

```
SC_MODULE(com_buffer_class) {
   // Ports
   sc_in< sc_logic > clk;
   sc_in< sc_logic > rst_a;
   sc_in< sc_logic > dready_out;
   sc_in< sc_lv<12> > dout_out;
   sc_in< sc_lv<7> > adout_out;
   sc_out< sc_logic > daccept_out;
   sc_in< sc_logic > dreq_in;
   sc_in< sc_lv<7> > adout_in;
   sc_out< sc_lv<12> > d_in;
   sc_out< sc_logic > davail_in;

   void process_clk();

   SC_CTOR(com_buffer_class) {
      SC_METHOD(process_clk);
      sensitive << clk;
      SC_METHOD(next_state);
      sensitive << rst_a;
   }
   ~com_buffer_class();
   void next_state();
private:
   sc_lv<12> ComBuffer[136];
   int block_read_p;
   int block_write_p;
   int element_write_counter;
   int element_read_counter;
};
```

The module has two processes. The first process is implemented in the function *process_clk*. This process is sensitive to the *clk* input and when the clock is high, it allows reading and writing to the communication channel. The code used therefore is the same as is used in the C model for the communication channel. The A|RT data types have only been converted to SystemC data types.
The second process is implemented in the function *next_state*. This process is sensitive to the *rst_a* input. This function resets the communication channel when the reset is high.

The communication RAM arbiter has been implemented in the following module:

```
SC_MODULE(ram_arbiter_class) {
   // Ports
   sc_in< sc_logic > clk;
   sc_in< sc_lv<1> > rw_port_1;
   sc_in< sc_lv<6> > a_port_1;
   sc_in< sc_lv<16> > d_port_1;
   sc_out< sc_lv<16> > dout_port_1;
   sc_in< sc_lv<1> > rw_port_2;
   sc_in< sc_lv<6> > a_port_2;
   sc_in< sc_lv<16> > d_port_2;
   sc_out< sc_lv<16> > dout_port_2;

   void process_readwrite();

   SC_CTOR(ram_arbiter_class) {
      SC_METHOD(process_readwrite);
      sensitive << rw_port_1;
        sensitive << rw_port_2;
        sensitive << a_port_1;
        sensitive << a_port_2;
   }
   ~ram_arbiter_class();
   void next_state();
private:
   sc_lv<16> ComRam[50];
   ifstream f_var0;
};
```

The module has one process. This process is implemented in the function *process_readwrite*. This process is sensitive to the *rw_port_1*, *rw_port_2*, *a_port_1* and *a_port_2* input. If the process is activated and the clock is high, it allows reading and writing to the shared memory. The code used therefore is the same as is used in the C model for the communication RAM arbiter. The A|RT data types have only been converted to SystemC data types.

The process in the *ram_arbiter_class* module is sensitive to a large number of signals. The module must be sensitive to its *rw* (read/write) and *a* (address) input on both sides of the module, because we must be able to handle a number of situations correctly. First is it possible that the *rw* signal changes and we must be able to act on that. It is however also possible that the *rw* signal does not change. This is the case if we perform multiple read or writes after each other. When this occurs only the address signal changes and the *rw* signal does not. To handle this situation, we must make the process sensitive to the address signal as well.

### 10.2.4. Bit stream processor

A SystemC model is generated with A|RT for the bit stream processor using the same architecture and C input as with the VHDL model.

There are however a number of problems with the A|RT SystemC output. Let us first discuss the problems involved with the class *artd_bitsp* (see files artd_bitsp.cxx and artd_bitsp.h of the A|RT output). This class keeps the *ram_image_data* block inside. This block is used to share data with the image processor in the JPEG decoder. It should therefore be made external to this A|RT processor. This implies that a reference to a class that implements this block should be provided to the constructor of the class *artd_bitsp*. This referenced *ram_image_data* block should then used instead of a locally declared *ram_image_data* block.

A second problem involves the *bitasu_1* block. This block provides access to the compressed image data stream. The bit ASU must therefore have access to this compressed image data stream. In the functional description of the bit stream processor, was defined that this access is implemented via an input block (see paragraph 8.1.2). An instance of the C++ class that implements this input block input block must be inserted into the class *artd_bitsp*. This requires that we add an instantiate *input block* from the type *Ctc_inputblock* and an instance *inputblock_out* from the type *Ctc_input block_out* to the class *artd_bitsp*. The constructor for the class *artd_bitsp* should be changed, such that a reference to an implementation of the class *inputblock_out* is passed to the class *artd_bitsp*.

The design then has an input block, but this block is never scheduled. The input block must be scheduled every cycle. This can be done by calling the *cycle_start*, *cycle_end* and *next_state* function of the input block whenever the run function of the *artd_bitsp* class executes this stage of the cycle.

A fourth problem with the A|RT SystemC output is that it never schedules the default instruction of blocks, even if these blocks have side effects. (This is a bug in A|RT. - see Appendix A, problem 20). The bit ASU implementation however requires a call with the default instruction. This call sets the *read_io* output of the block to zero. This problem is solved by setting the *bitasu_1_read_io* registry to zero after the input block has processed the *read_io* value of the previous bit ASU call.

The changes that must be made to the *artd_bitsp* class can be found in Appendix F. These changes can be made to the A|RT SystemC output by running the systemc_adap script. This script can be made though the compilation of the file systemc_adap.cxx. This file can be found in the directory src/jpeg_pc2/systemc of the deliverable.

The class *artd_bitsp* that is obtained after these changes are made must now be embedded into a SystemC module. The general set up of a SystemC module is described in the previous paragraph. A|RT creates such a block. But this A|RT SystemC module for the bit stream processor cannot be used. It does not contain implementations for the class *inputblock_out* and the class *ram_image_data* block. In stead the SystemC module in the files artd_bitsp_module.cxx and artd_bitsp_module.h in the directory src/jpeg_pc2/systemc of the deliverable should be used.

The interface of the SystemC ar*td_bitsp_module* is shown in Figure 10-12.

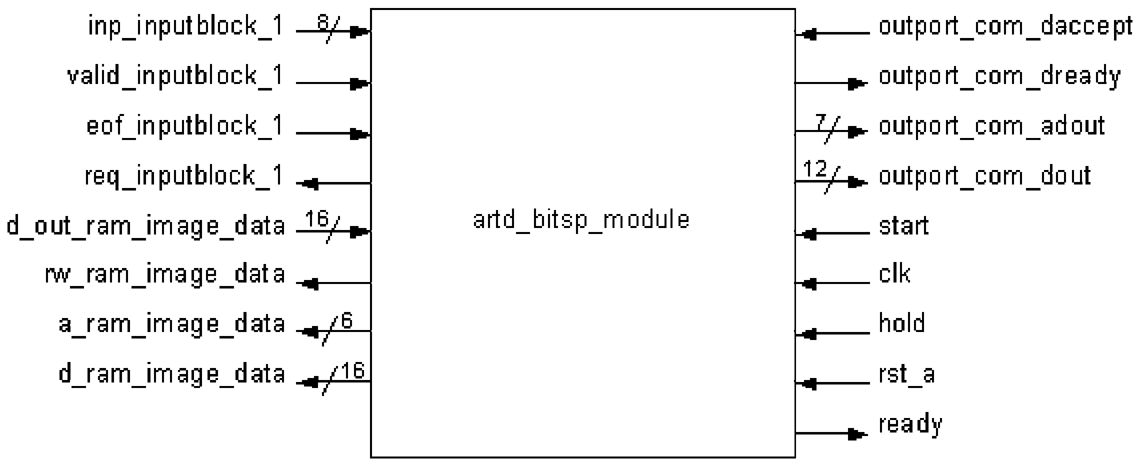


**Figure 10-12 Interface of SystemC module artd_bitsp_module**

### 10.2.5. Image processor

A SystemC model is generated with A|RT for the image processor using the same architecture and C input as with the VHDL model.

There are a number of problems with this A|RT SystemC output. Let us first discuss the problems involved with the class *artd_imagep* (see files artd_imagep.cxx and artd_imagep.h of the A|RT output). This class keeps the *ram_image_data* block inside, just as the A|RT SystemC output of the bit stream processor did. The solution to this problem is also same as with the bit stream processor.

The changes that must be made to the class *artd_imagep* can be found in Appendix F. These changes can be made to the A|RT SystemC output by running the systemc_adap script. This script can be made though the compilation of the file systemc_adap.cxx. This file can be found in the directory src/jpeg_pc2/systemc of the deliverable.

A|RT generates also a SystemC module in which the A|RT processor is incorporated. This module can however not be used, as it does not contain an implementations for the class *ram_image_data* that implments this block. Instead of using the SystemC module outputted by A|RT, the SystemC module in the files artd_imagep_module.cxx and artd_imagep_module.h in the directory src/jpeg_pc2/systemc of the deliverable should be used.

The interface of the module artd_imagep_module is shown in Figure 10-13.

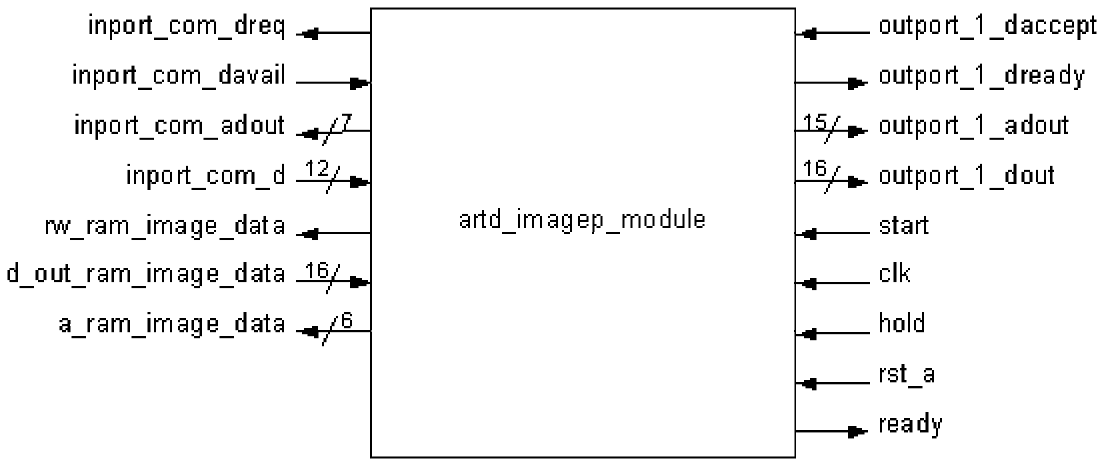


**Figure 10-13 Interface of SystemC module artd_imagep_module**

### 10.2.6. JPEG decoder

The JPEG decoder can be constructed using the SystemC modules of the communication RAM arbiter, communication channel, bit stream processor and image processor. Figure 10-14 shows how the ports of the different modules should be connected.

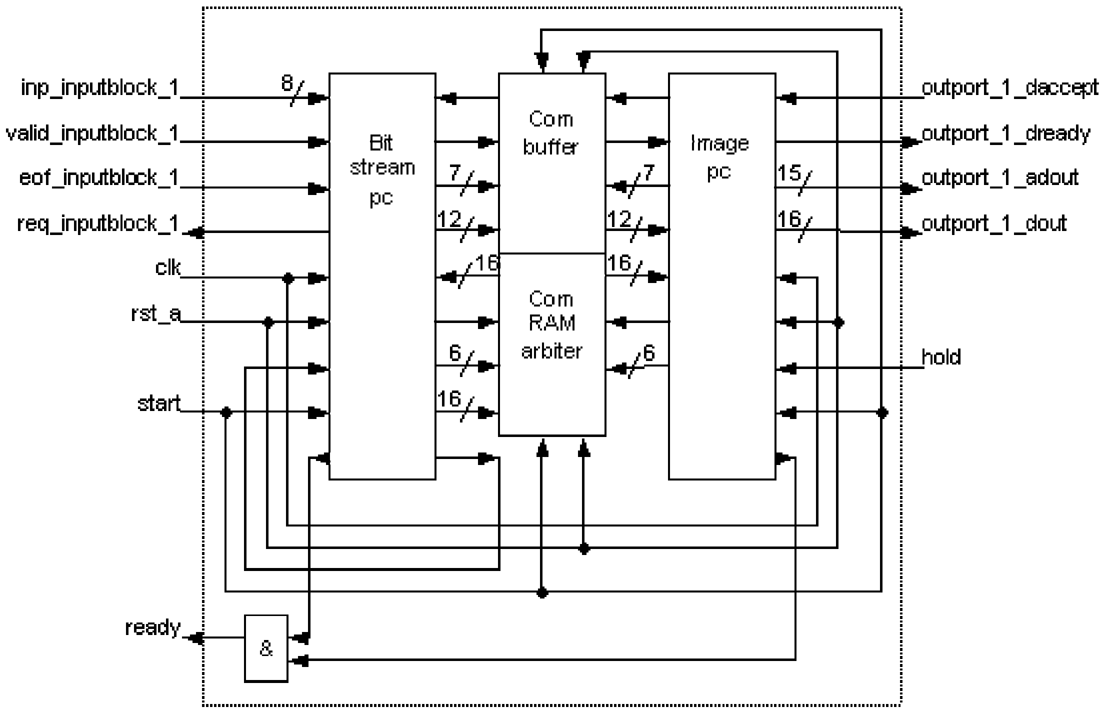


**Figure 10-14 Implementation of JPEG decoder**

## *10.3. Test bench*

The previous paragraph described the SystemC model for the bit stream processor, image processor and the JPEG decoder. To prove that the models are correct, a set of test benches has been made. This set contains a bench for the bit stream processor (bitsp_module_bench.cxx), a bench for the image processor (imagep_module_bench.cxx) and a bench for the JPEG decoder (jpeg_module_bench.cxx). These test benches can be found in the deliverable under the directory src/jpeg_pc2/systemc.

The input and reference files that are supplied for every bench are the same as with the VHDL test bench.

The architecture used for the JPEG decoder test bench is shown in Figure 10-15.

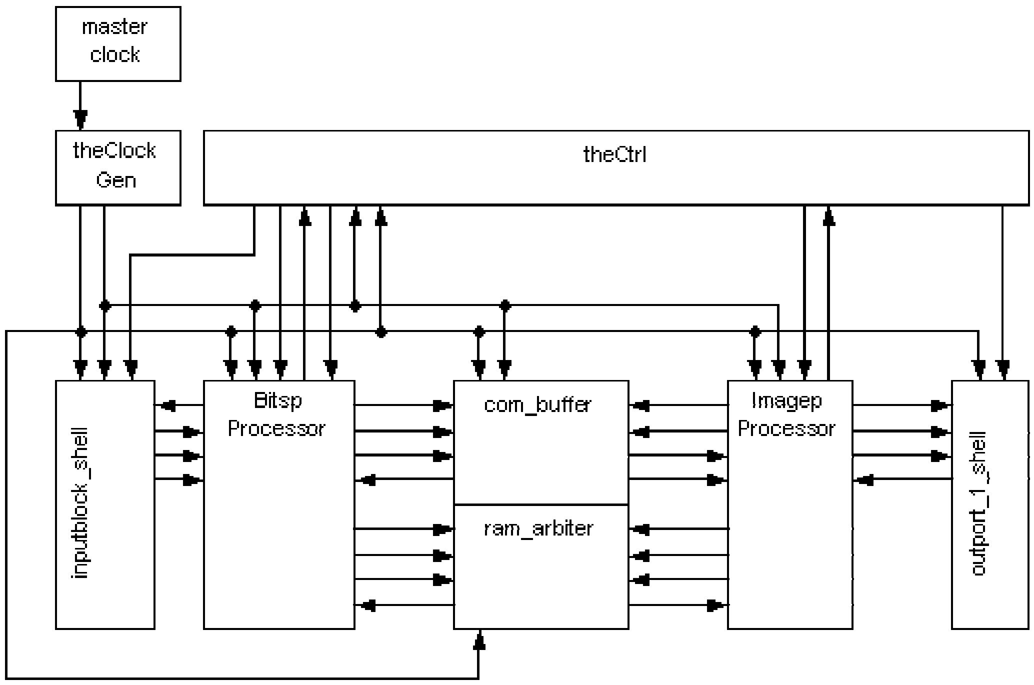


**Figure 10-15 Architecture of SystemC bench for JPEG decoder**

## *10.4. Experimental results*

Using the test benches described in the previous paragraph, simulations of the bit stream, image processor and JPEG decoder have been made. The experimental results obtained from these simulations are presented in this paragraph.

Before we do so, we want to make two remarks on the SystemC libraries (version 1.0.1 and 2.0) we used during the simulations. During these simulations, we noticed an extreme memory usage (up to 1GByte) by the executables. This turned out to be caused by a memory leak in the range object of the SystemC library. Due to this memory leak is it only possible to simulate using small images (less then 30000 pixels).
We found a second problem with SystemC when we removed the clock signal from the shared memory. This allows writes to the memory when the clock signal is low. We knew that in the simulation situations occur in which the clock is low, the write signal is active and the data present at the input is incorrect. Thus, as a result of that we were expecting some problems with the simulation. When using SystemC library 1.0.1 however, nothing happened and the whole simulation was correct, while it in fact was supposed to fail. The reason that the simulation went right was found in the fact that the SystemC scheduler used in the simulation did not schedule the SystemC module that implements the shared memory when the clock is low. This seams to be a bug in SystemC 1.0.1. The SystemC library 2.0 did not have this problem. It scheduled the SystemC module that implements the shared memory in the above-described case.

The bit stream, image processor and JPEG decoder have been simulated using the three test benches described in the previous paragraph. The simulation was done by compiling the test bench source files and then running the tests.

The most important result from the tests is that the SystemC models for the bit stream, image processor and JPEG decoder produce outputs that not differ from the reference files. From this we may conclude that the SystemC models are correct.

With the simulations, we obtained also some performance characteristics on the designs. These characteristics are presented in Table 10-2. The first column contains the design that was simulated. The second column lists the number of cycles needed to perform the actions of the design under test on the compressed image data. The third column lists the number of cycles needed to initialize the design under test. The last column lists the time needed to run the complete simulation (decompression of one image).

**Table 10-2 Performance characteristics**

| Design | #Cycles/image | #Cycles init | Time/image [min:sec] |
|---|---|---|---|
| Bit stream | 368501 | 2 | 0:32 |
| Image | 528463 | 0 | 0:28 |
| JPEG decoder | 676574 | 0 | 1:15 |

Comparing the performance characteristics of the simulation of the design using SystemC (see Table 10-2) and VHDL (see Table 9-1) leads to the observation that the number of cycles needed by both models to compute the outcome differs. This implies that the VHDL and SystemC model do not completely match at cycle-accurate level. The reason for this difference has not been studied. Therefore, no comment can be made on the cause of this difference.

When we defined our JPEG decoder architecture, which uses a multi-processor system, we declared that both subsystems should require around the same system load. The simulation results indicate that the bit stream processor needs around four hundred thousand cycles and the image processor needs approximately five hundred thousand cycles. This implies that when the image processor is fully occupied, the bit stream processor is only used at 70% of its capacity. We can conclude from this that we didn' t find a subdivision in which both subsystems have a load of 100%. However, we did find a reasonable match between the system loads of the two subsystems if we bear in mind that we made the division into the two subsystems on very loose assumptions.

A third observation can be made when comparing the time needed for the simulation in SystemC and VHDL. The SystemC simulation is around 20 times faster as the VHDL simulation, while the resulting information from the simulation is equal.
This time benefit can be exploited to make a number of simulations of a system to obtain optimal settings.

A fourth observation can be made if we look at the number of cycles needed by the JPEG decoder to decode an image. It is considerably larger as the number of cycles needed by the slowest subsystem (image processor) to perform its operations on the data. A part of this difference is caused by the fact that the image processor must wait a number of cycles before the bit stream processor produces its first communication packet. However, most of he delay is caused by the fact that the two processors must wait for each other before they get access to the communication channel. The communication channel forms the bottleneck in the system.

Our goal is now to remove this bottleneck. We have therefore made a number of simulations with different values for the number of communication packets that can be stored in the communication channel. The results of these simulations are shown in Table 10-3.

**Table 10-3 Performance of JPEG decoder using different channel lengths**

| #Packets in channel | #Cycles/image | #Init cycles |
|---|---|---|
| 1 | 804635 | 0 |
| 2 | 676574 | 0 |
| 3 | 629618 | 0 |
| 4 | 600714 | 0 |
| 5 | 596556 | 0 |
| 6 | 592452 | 0 |
| 7 | 578783 | 0 |
| 8 | 570949 | 0 |
| 9 | 570949 | 0 |
| 10 | 570949 | 0 |

The table indicates that when the communication channel can store eight or more communication packets at a time the decoder won' t operate faster. This leads to the conclusion that when we use a communication channel that can store eight communication packets at a time, we obtain a decoder in which one of the two processors forms the bottleneck and not the communication channel.

The final JPEG decoder design thus needs 570949 cycles to decode a JPEG file of 15kbytes. If we assume that a clock frequency of 40MHz is used for the JPEG decoder, we obtain a data rate of 1,03 Mbytes/s for the decoder.

# 11. Conclusion and recommendations

This report describes the design and implementation of a JPEG decoder. A multi-processor architecture is used to implement the system in C, VHDL and SystemC. The development of the JPEG decoder was done with the following goals:

- Design a multi-processor JPEG decoder.
- Implement designed JPEG decoder in VHDL and SystemC.
- Provide Adelante Technologies with feedback about their SystemC integration.
- Provide the TU Eindhoven with suggestion on changes that can be made in the academic course "Design of large scale integrated circuits".

To summarize the outcome of the project, this chapter first describes the realized goals, and then it gives some recommendations for future developments.

## *11.1. Conclusion*

In chapter 3, we described a general architecture for a multi-processor JPEG decoder. This formed the basis for our design process. This architecture was comprised of two processors (bit stream and image). A communication protocol was defined in chapter 4 to enable communication between these processors.

In the chapters 5, 6 and 7, we introduced a functional description of the bits stream, image processor and communication logic. All systems were disassembled into a number of functional units. We then showed the relationship between the units and subsequently described the function of all these units.

With the description of the bit ASU and the input block in the chapters 5.5 and 5.6, we introduced two units who facilitate the reading from the compressed image data stream. These units make an efficient decoding process possible, because most bit operations are incorporated in them. The bit ASU further enables the possibility to create a fast control flow by splitting markers, which define the structure of the compressed image data, into flags that can be handled by the systems controller.

In chapter 8, a C model was presented that implements the described JPEG decoder. With this C model, we could demonstrate that the design is a valid JPEG decoder.

In the chapters 9 and 10, we presented a VHDL and SystemC model for the design. This was done with the A|RT tools and a system architecture for both processor that is comprised of mostly standard functional blocks. In chapter 10.4, we presented some experimental results obtained from a simulation of the SystemC model. We used these results to optimize the communication channel and made some remarks on the design. The first remark is about the observation that the two processors in the design do not have the same load. We concluded from this that we did not find a subdivision in which both subsystems have a load of 100%. However, we must conclude that we did find a reasonable match between the system loads of the two subsystems if we bear in mind that we made the division into the two subsystems on very loose assumptions.

Another important remark about the design its about its speed. The designed JPEG decoder works with a throughput of around 1,3 Mbytes/s, which is slow. This is mainly caused by three reasons. First, the subsystems use standard functional block. These blocks are not optimized for the JPEG decoder. They are therefore slow compared to optimized blocks. The second reason is that large parts of the JPEG decoder algorithm are not optimized. The algorithm for instance contains a number of loops in which the loop condition can be modified in the loop body. This disables the possibility of loop folding and thus of a fast implementation. The decoding speed is also negatively influenced by the fact that the color conversion and re-ordering procedures are embedded in the design. Most JPEG decoder do not have these blocks inside them and can thus operate faster.

In general, we can state that we have fulfilled both the first and second goal with the results presented in this report. The third and fourth goal will be fulfilled by the recommendations made in the following paragraph.

## *11.2. Recommendations*

In this report, we described the design and implementation of a JPEG decoder. The resulting JPEG decoder implementation was simulated using SystemC and VHDL. These simulations proofed that the design functions according to our expectations. However, the design is far from optimal and further improvements are possible. In this paragraph, we will formulate recommendations and possibilities for improvements on the design. We will further formulate recommendations on improvements in the A|RT tools, the SystemC implementation of Adelante Technologies and on changes that can be made in the academic course "Design of large scale integrated circuits".

The communication logic consists of a communication channel and a communication RAM arbiter. The last unit is needed to give the image processor access to image data (height, width etc.). The image processor accesses this memory when it has processed the first block. The bit stream processor has already initialised the values for all variables. Both processors will not change these values anymore. This property opens the possibility to remove the communication RAM arbiter from the design. This can be done by introducing a set-up channel packet and giving both the image and bit stream processor a memory to store the image data. As soon as the bit stream processor has initialised all variables in its memory sends it a set-up channel packet that contains the values for these variables to the image processor.

A second possibility for improvement of the JPEG decoder can be found in splitting the image processor in two subsystems. The image processor now consists of an IDCT process that consumes and produces data at one block at a time. The second process processes the MCU (colour conversion and re-ordering) consumes up to ten blocks at a time and produces one image. The data consumption/production pattern of the IDCT process and process that processes the MCU do not match. Splitting it into two systems can solve this. Both systems can then be optimised to their own task and data production/consumption rate.

A third option to obtain improvements in the JPEG decoder is to look for optimisations of the algorithms and the functional blocks in the processors. The colour conversion and re-ordering unit for instance use complex algorithms for the address calculation. These address calculations can probably be modified to do calculations that are less time consuming. It will further be possible to incorporate large parts of these calculations in an ASU.

In chapter 10.1, we discussed the A|RT SystemC output and the timing model used to simulate one cycle of the A|RT processor. The A|RT SystemC output consists of a test bench

that uses SystemC modules. Processes in these modules are activated whenever a signal changes for which they are sensitive (e.g. clock) changes. These processes consist of lines of C++ code that are executed in sequence. Therefore, the execution of a process is not more then running a standard C++ function.
The process inside the SystemC module generated for the A|RT processor executes thus all functional units in sequence and not, as in VHDL, based on their sensitivity to signal changes. This makes that the process inside the SystemC of the A|RT processor is equivalent to a standard instruction set simulator that simulates the design for one cycle.
The results obtained from the SystemC simulation should therefore be seen as an indication that the functional behaviour of the design is correct at the instruction set level.

It is however possible to generate an A|RT SystemC output that simulates at the RT level (uses the concept of delta cycles). The A|RT processor should therefore be implemented in a SystemC module that contains instantiations of SystemC modules for all blocks in the design. This adds an extra hierarchical level to the design and makes the layout and behaviour comparable to VHDL. The simulation is then no longer an instruction set simulation encapsulated in a SystemC bench. Another advantage of this modification to the A|RT SystemC output is the fact that the timing model used in the A|RT processor is no longer needed. This is because every SystemC module is executed in the bench as soon as a process request data. When for instance, a SystemC module that implements an input in the A|RT processor is asked for data, a process in this module is executed. This results in sending out a signal to the bench that the input module needs data. Then a process in the SystemC module in the bench that has to supply this data is executed. This module set the data on its output. Thereby it changes a port for which the input is sensitive. A process in the SystemC module for the input is executed. This process can then place the data on the output of the module, so that the SystemC module that requested can use it. This all happens in a number of delta cycles inside one instruction cycle.
There is however one disadvantage to this solution and that is that when the A|RT processor is split in a number of different SystemC modules, all these modules have to be scheduled by the SystemC scheduler. A simulation in which this scheduler is extensively used is of course slower then a simulation in which mainly sequential C code is executed.

It is recommended that considering the above mentioned advantages and disadvantages of splitting the A|RT SystemC output in multiple SystemC modules, further study is made of the consequences of the idea to split the A|RT processor. This study can then be used to make a well-founded decision whether or not this should be done.

A second recommendation on the A|RT SystemC output is about documentation. The timing model used in the A|RT SystemC implementation is rather complex to understand. This model must be understood, if an application specific unit is used that uses inputs or outputs. This was the case with the input block. Therefore the timing model used in A|RT SystemC was explained several times during this traineeship. These explanations seemed to contradict. This was caused by the fact that in the different explanations used different definitions for the words input, output and state. The word input was for instance once used exclusively for inputs from the outside world, while another time it was used for inputs to the functional block that originated inside the processor.
To prevent confusion about these definitions, one should document these definitions and the timing model carefully and then only use these definitions.

A third recommendation on the SystemC A|RT output is about the fact that functional blocks cannot be placed outside the C++ class that models the A|RT processor. When A|RT generates VHDL a block can be made external by using a pragma. This pragma is not supported by A|RT when it generates SystemC output. To support cases in which a designer wants to place functional blocks outside the design, A|RT should support this pragma. Else, the designer is forced to make these changes, which is a time consuming task.

The functional block that implements the RAM is in the A|RT SystemC output not a C++ class which implements this functional block as is every other functional block. The RAM is implemented in the A|RT SystemC output as an array. Reading and writing to the memory is implemented as direct accesses to this array. This implementation makes it difficult to make the memory external and it does not follow the rules defined by A|RT for the implementation of functional blocks in the A|RT SystemC output. It is therefore recommended that the implemented of the RAM is not an array, but as a C++ class as is any other functional block. This makes the A|RT SystemC output for the A|RT processor uniform.

This report describes the design of a multi-processor implementation of a JPEG decoder. The general design of such a JPEG decoder is discussed in the course "Design of large scale integrated circuits". A part of the decoder algorithm is also implemented in a C model. A system architecture for this part of the decoder is developed and optimised using A|RT Designer.
This design process ends at the scheduling level of A|RT. Generation of a VHDL model will probably be too difficult for students who are not familiar with VHDL. However, the experience of simulating at the cycle-accurate level gives a good inside in the way processors operate. With the possibility of generating SystemC with A|RT the problem of the lake of knowledge of VHDL can be overcome as SystemC requires only knowledge of C++. Most students are familiar with this language. A basic introduction into the concepts in SystemC will then be enough to let students simulate at a cycle-accurate level.
With the possibility to simulate a design it might also be attractive to create application specific units and use those in the design and simulation. Students can then experience what a cycle-accurate algorithm looks like and how it can be simulated.

# References

[1] International Telecommunications Union, *Information technology – Digital compression and coding of continues-tone still images – Requirements and guidelines (Recommendation T.81)*

[2] Stuijk, S., *VLSI design van een bittrue implementatie van IDCT algoritme (project report course " Design of large scale integrated circuits - 5P140", TU Eindhoven, 2001)*

[3] Synopsys Inc., *SystemC Version 1.1 User'  s Guide (2000), http://www.systemc.org*

## Appendices

# Appendix A Problems

This appendix contains a list of bugs found during the implementation of the JPEG decoder. For every bug a description of the problem is given. This description includes the tool name and version number with which the bug was found. In addition, an ID number is given with every problem; this is the defect number under which Adelante Technologies knows the problem.

**Problem 1**

| | |
|---|---|
| ID | DR203977 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev7 |
| Description | It is possible that the scheduler enters an eternal loop during the tracing of a unique name for variables that are located in registry file. |

**Problem 2**

| | |
|---|---|
| ID | DR203973 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev7 |
| Description | If a global variable is used in a loop condition, then this loop is converted to a do-loop. Even if the global variable cannot be modified in the loop. |

Example:
```
for (IT i=0; i<globalVar; i++) {
```

A workaround is to use the following code:

```
IT tmp(globalVar)
for (IT i=0; i<tmp;i++) {
```

**Problem 3**

| | |
|---|---|
| ID | DR204000 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev7 |
| Description | In some cases A\|RT Designer may fail to read to sfg in the architecture creation step. |

**Problem 4**

| | |
|---|---|
| ID | DR204002 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev5 |
| Description | If the C function shown below is used without the pragma, the mapping fails with the following message: |

*Cannot continue mapping to architecture due to wrong usage of global or static variable ' bitcount' in function jpeg.*

```
Uint<16> get_bits(const Uint<8> number, Uint<1>& ErrorBitASU)
{
#pragma OUT ErrorBitASU
      Uint<16> result = 0;
      Uint<8> i = 0;

      for (i=0; i<number; i++) {
            result = (result << 1);
            result[0] = get_one_bit(ErrorBitASU);
      }

      return result;
}
```

There are two problems with this error message:

1. The variable bitcount is not used in the function jpeg but in the function get_bit. The test should thus be performed BEFRORE function expansion or this information kept during function expansion

2. A normal user is NOT able to correlate this error message to the actual error in the source code. You have to know that it is possible caused by an unmapped pragma in the source.

Conclusion: A map_function pragma that is not mapped must give a warning.

**Problem 5**

| | |
|---|---|
| ID | DR204003 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev5 |
| Description | The mapping of the C function shown below fails. It cannot pass two flags on the ALU. This is correct. However, those flags need to be mapped on the controller. |

```
void function(Uint<1>& bit0, Uint<1>& bit1)
{
   ReadMarker(bit0,bit1);

   do {
     if (!bit0 && bit1) {
       Action_1();
       ReadMarker(bit0,bit1);
     }
     else {
       Action_2();
       ReadMarker(bit0,bit1);
     }
   } while(!bit1 && bit0);
}
```

**Problem 6**

| | |
|---|---|
| ID | DR204005 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev7 |
| Description | If no_connection pragmas are added in the architecture pragma file to remove the connections that cause the previous problem, the mapper stops with the following message: |

*There is not path to route the result of operation.... to destination....*

There are three problems with this message:

Problem 1:
The operation has about 10 outputs. It would be nice to know which output cannot be routed.

Problem 2:
Source reference of destination is missing.

Problem 3:
A|RT should not try to map the flag on the ALU and use the unfound connection, but A|RT should map the flags on the controller (see previous problem).

**Problem 7**

| | |
|---|---|
| ID | DR204020 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev5 |
| Description | A function that contains a loop with loop initialization is sometimes incorrectly initialized. The jump logic jump to the execution of the loop instead of its initialization. |

**Problem 8**

| | |
|---|---|
| ID | DR204035 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | A function that contains a for-loop with loop initialization and who is called several times inside a while loop is incorrectly initialized. The jump logic jump to the execution of the for-loop instead of its initialization.<br><br>A workaround is to manually unroll the for-loop. |

**Problem 9**

| | |
|---|---|
| ID | DR204042 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | In the case that a design accesses dead/semi-dead global variables, the mapping step gives the following error:<br><br>*Operation '  /name of main function/'   cannot be mapped (no map_expression or map_function matched). No resource found capable of implementing it.* |

**Problem 10**

| | |
|---|---|
| ID | DR204043 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | A write-only, global array cannot be mapped on an output. The mapper ignores mapping rules that request this by saying that local variables can |

only be mapped on a RAM.

A workaround is to include the array in the function header of the main function and pass this array to all functions that want to use it. This array can then be mapped onto an output.

**Problem 11**

| | |
|---|---|
| ID | DR204044 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | Structures that are only used for writing are thrown away. Even if these structures are declared volatile. |

The following workaround can be used. Given a struct:

```
struct rsize {
    int x;
    int y;
};
```

This should be rewritten into:

```
int rsize_x;
int rsize_y;
```

**Problem 12**

| | |
|---|---|
| ID | DR204046 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | A segmentation violation at the beginning of the scheduling step can be encountered. |

This problem is related to the next problem.

**Problem 13**

| | |
|---|---|
| ID | DR204047 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | It is possible that in the mapping step the tool hangs after mux generation. The memory consumption goes straight up until 600 Mbytes and then it finished normally. The memory leak can be fixed by switching to Perl 5.6.1. |

**Problem 14**

| | |
|---|---|
| ID | DR204049 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | The net listing generates a segmentation violation. This happens with CTC on not with VHDL and is caused by the writing of don' t care to RAM. |

**Problem 15**

| | |
|---|---|
| ID | DR204050 |

| | |
|---|---|
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev5 |
| Description | The ProcessFrameHeader() function starts with function calls to the ReadSegmentSize() and ReadByte() functions. The execution of these functions is scheduled at the end of the ProcessFrameHeader() function execution. While these functions have side effects and should therefore be sequenced. |

**Problem 16**

| | |
|---|---|
| ID | DR204051 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | The net listing step takes 15 minutes. It hangs on the step ' making components external' . This transformation should not take that much time. |

**Problem 17**

| | |
|---|---|
| ID | DR204057 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev8 |
| Description | The JPEG decoder design consists of two processors. If in the ProcessBlock() routine, the element values of the block are calculated in a temporary array T and then copied to the output port. Then, all values of the array T are always zero. This is because the processor jumps too early out of the loops that calculates these values. |

**Problem 18**

ID DR204067
Tool A|RT Designer Pro
Version v2.4 rev9
Description If a while loop with the following C statement is used:

```
while(ComOut[0] != 0xFFF)
```

and ComOut is an input (even declared volatile).

Then the input port is read only once. This can be seen in the schedule report. This input port should be read every time the loop condition is evaluated.

**Problem 19**

ID DR204097
Tool A|RT Designer Pro
Version -
Description The rules in C simulation are

```
cycle_start  output = f(state)
cycle_end   next_state = f(fstate,input)
update       if (!hold) state = next_state
```

For most blocks cyle_start and cycle_end can be merged since there are no external inputs.

In the JPEG decoder, the input block ASU performs external I/O. The optimization of merging cycle_start and cycle_end can thus not be performed.

**Problem 20**

| | |
|---|---|
| ID | DR204099 |
| Tool | A\|RT Designer Pro |
| Version | v2.4 rev9 |
| Description | A default instruction with side effect is not scheduled in cycle true C. |

# Appendix B JPEG decoder operations

In this appendix a describition of the transformation involved in the JPEG decoding process are given.

## B.1 Inverse discrete cosine transform

Using the discrete cosine transform it is possible to go from the space domain to the frequency domain. Using the inverse discrete cosine transform the data undergoes the opposite transformation. This transformation gives the possibility to go from the frequency domain to the space domain.
The inverse discrete cosine transform for a block of 8 by 8 pixels is given by the following equation:

$$X(i,j) = \sum_{k=0}^{7}\sum_{l=0}^{7} \frac{c(k)c(l)}{4} Y(k,l) \cos\left(\frac{(2i+1)k\pi}{16}\right) \cos\left(\frac{(2j+1)l\pi}{16}\right) \qquad \text{(B-1)}$$

c(k) is given by:

$$c(k) = \begin{cases} 1/\sqrt{2} \ if\ k = 0 \\ 1 \quad else \end{cases}$$

## B.2 Dequantisation

During the encoding process every element of a block S has been quantised. This means that every element of the block S was divided by a value selected for its postion in the block and subsequently rounded to the nearest integer. By doing this, the encoder is able to reduce the amount of information present in the block. This process does not greatly influence the quality of the image but realizes considerable data reduction.

During the dequantisation. Every element of the block S is multiplied by the value that it was divided by during the encoding process. This quantisation value can be found in the quantisation table Q at position (v,u). For a element (v,u) of the block S the dequantization is then given by the following equation.

$$R_{vu} = S_{q_{vu}} * Q_{vu} \qquad \text{(B-2)}$$

## B.3 Zig-Zag scan

During the quantization a large number of coefficients is rounded to zero. To encode large runs of zeros, JPEG uses run-length encoding [RLE]. To exploit the benefits of RLE the encoder reorders the elements in the block according to Figure B-1. In this way, the block is ordered to the raising frequency. The low frequencies (white) are placed in front; they are followed by the middle frequency components (light gray). The last elements of the block are then the high frequencies (dark gray). This gives the highest change of having a large row of zeros.

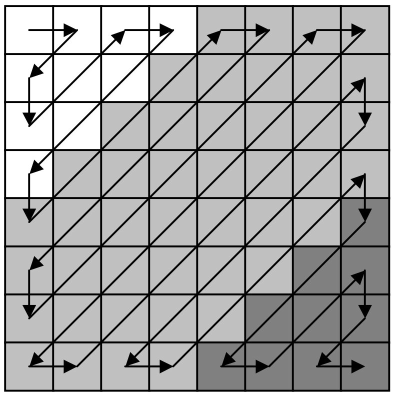


**Figure B-1 zig-zag scan**

The Zig-Zag scan in the decoder must reorder the elements of the block in the opposite way, such that the elements are placed into their original position in the block.

## B.4 Variable length decoding

In the variable length decoder, the compressed image data is uncompressed. This is done using a standard Huffman decoding algorithm.

The first element to be decoded is called the DC element. This element is placed at position (0,0) in the block. For decompression of the symbol representing the DC element, a DC Huffman table is used.
The encoding process did however not encode the actual value of the element. Instead, it encodes the difference between this elements value and the value of the DC elements of the previous block. This difference calculation must also be undone by the decoder.

The other elements of the block are called the AC elements. The element values are decoded one after another by the Huffman decoder. It therefore uses an AC table.
Only one trick has been applied to the encoded bit stream. The encoder uses not only a Huffman encoder, but also a run length encoder. The decoding of a run length encoded symbol should be done before the bit stream is given to the Huffman decoder. To handle this properly, the byte stream must be monitored to see whether a part of de byte stream is run length encoded. This should then be decoded before the Huffman decoder receives its bits.

## B.5 Color conversion

The JPEG standard specifies that the output of the decoder, in case of a collored image, is of the form YCbCr. Most image display systems (e.g. video and computer) use a RGB coloring scheme. To change from the JPEG coloring scheme to the more often used RGB coloring scheme a color conversion must be used. This conversion is given by the following set of equations.

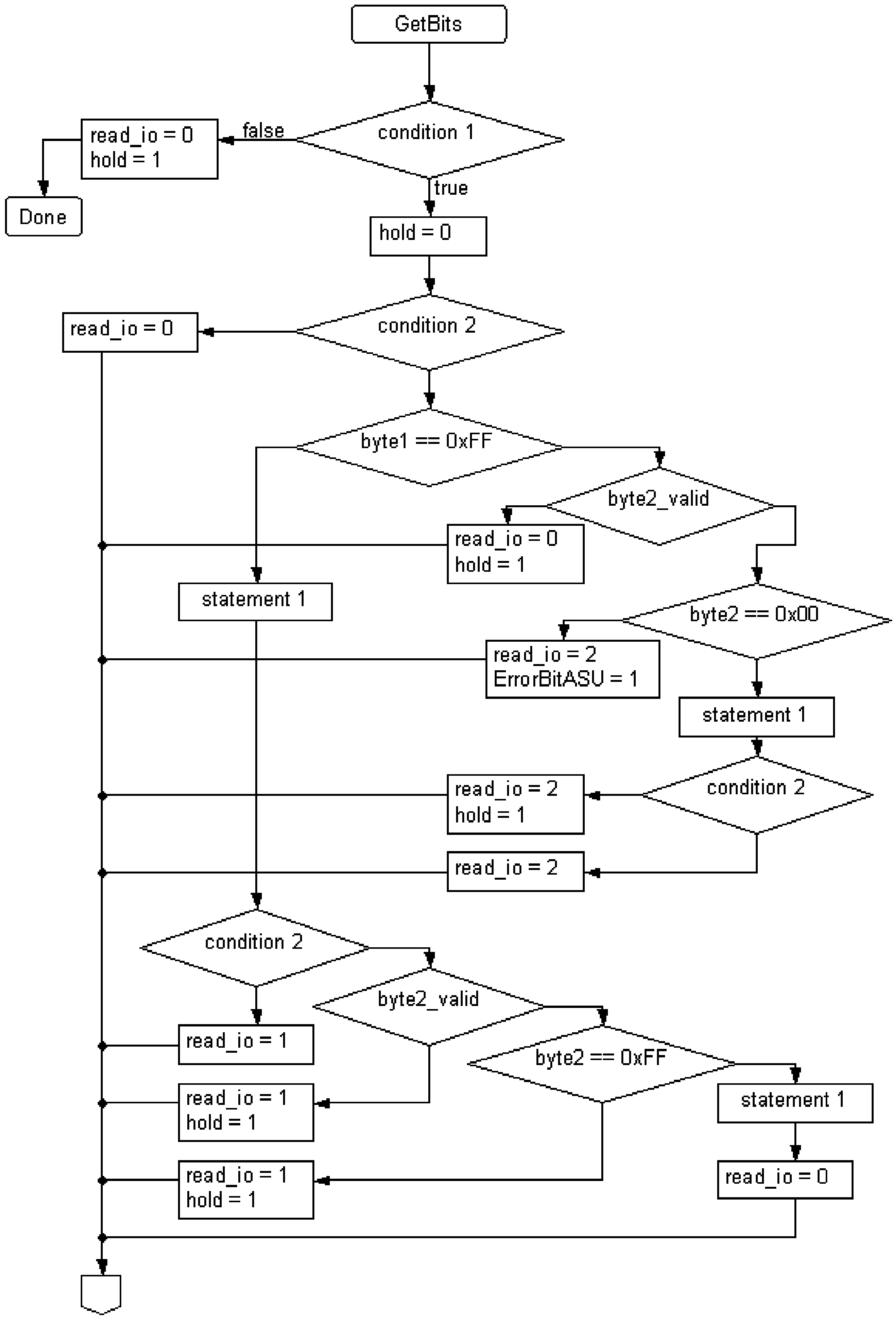
$$\begin{aligned} R &= Y + 1.402(Cr - 128) \\ G &= Y - 0.34414(Cb - 128) - 0.71414(Cr - 128) \\ B &= Y + 1.722(Cb - 128) \end{aligned} \qquad \text{(B-1)}$$

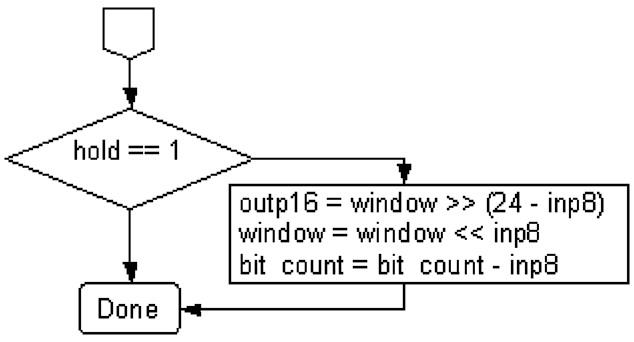
## B.6 Re-ordering

The color conversion unit outputs a set of blocks that are converted  converted from the YCbCr coloring scheme to the RGB coloring scheme. Such a set of blocks describes a region, JPEG calls this a MCU, of the image. Most image display systems use a line-by-line writing method. A line then start with the value of the first color component of the first pixel of the image. It is then followed by the value of the second color component of the image. This is repeated until the value of all color components of the first pixel are outputted. The system then outputs the values of all color components of the second pixel on the line in the same way. This is then done for all pixels on the line. When the line is finished, the system repeats this procedure for the next line, until the whole image has been outputted.

The re-ordering procedure converts the MCUs to the line-by-line output format.

# Appendix C Procedure GetBits





Condition 1:
(bitcount >= inp8) || (((bitcount + 8) >= inp8) && byte1_valid) || (((bitcount + 16) >= inp8) && byte1_valid && byte2_valid)

Condition 2:
bitcount < inp8

Statement 1:
window = window | (0xFF << (16 - bitcount))
bitcount += 8

**Figure C-1 Procedure for ReadMarker**

# Appendix D Procedure ReadMarker

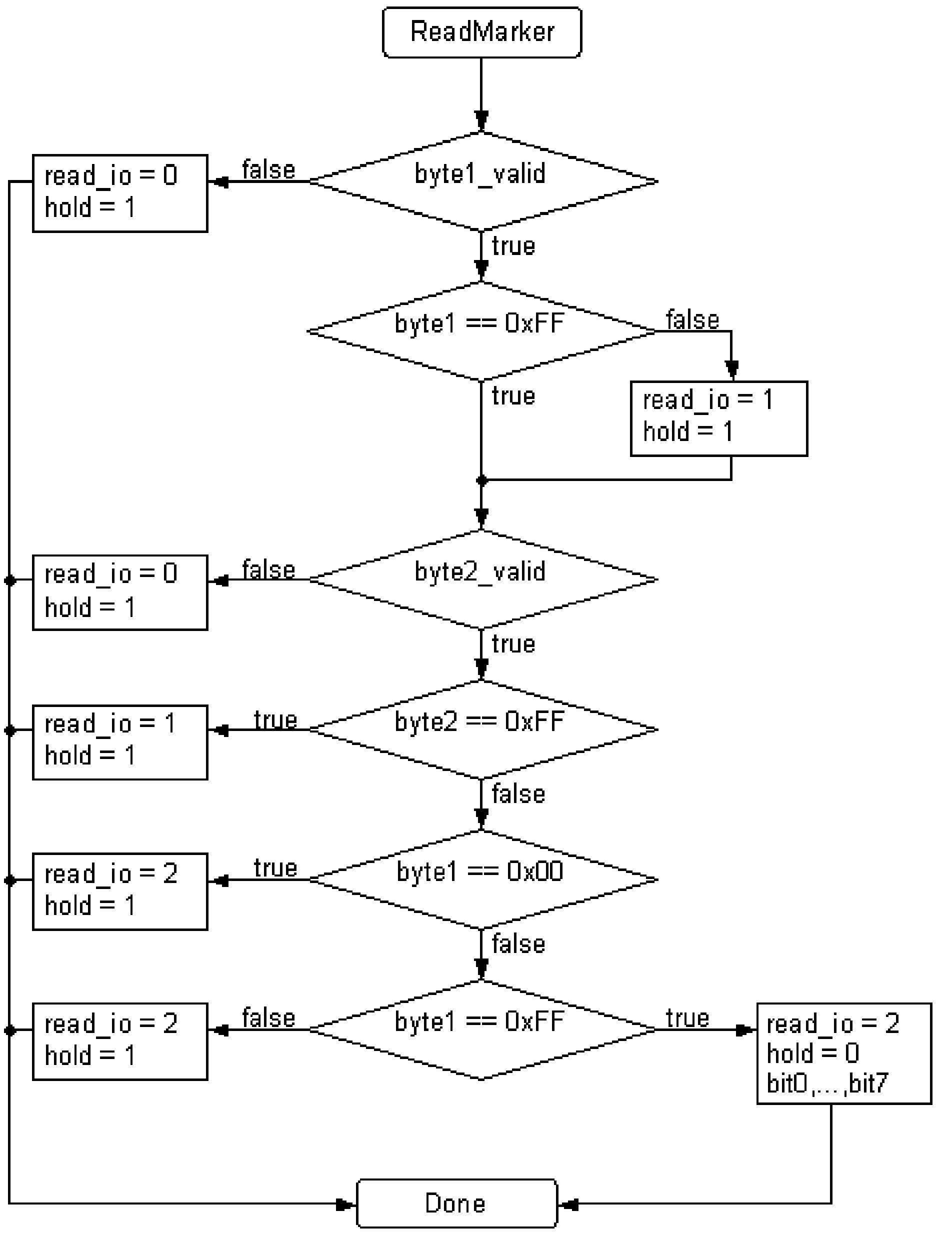


**Figure D-1 Procedure for ReadMarker**

# Appendix E Changes in VHDL model of bit stream processor

This appendix contains a list of changes that should be made to the VHDL generated by A|RT Designer for the bit stream processor.

**Modification 1**
Add to the entity declaration of the bitsp_processor the following code:

```
inp_input block_1 : in std_logic_vector( 7 downto 0);
valid_input block_1 : in std_logic_vector( 0 downto 0);
eof_input block_1 : in std_logic_vector( 0 downto 0);
req_input block_1 : out std_logic_vector(0 downto 0);
hold_bitasu_1 : out std_logic_vector(0 downto 0);
```

**Modification 2**
Add to the list of signals declared in the architecture of the bitsp_processor the following signals:

```
SIGNAL bus_read_input block_1 : std_logic_vector(1 downto 0);
SIGNAL bus_outp16_input block_1 : std_logic_vector(15 downto 0);
SIGNAL bus_valid_byte1_input block_1 : std_logic_vector(0 downto 0);
SIGNAL bus_valid_byte2_input block_1 : std_logic_vector(0 downto 0);
SIGNAL dup_input block_1_read : std_logic_vector(1 downto 0);
SIGNAL dup_input block_1_outp16 : std_logic_vector(15 downto 0);
SIGNAL dup_input block_1_valid_byte1 : std_logic_vector(0 downto 0);
SIGNAL dup_input block_1_valid_byte2 : std_logic_vector(0 downto 0);
SIGNAL dup_bitasu_1_read_io : std_logic_vector(1 downto 0);
SIGNAL dup_bitasu_1_io_outp : std_logic_vector(15 downto 0);
SIGNAL dup_bitasu_1_byte1_valid : std_logic_vector(0 downto 0);
SIGNAL dup_bitasu_1_byte2_valid : std_logic_vector(0 downto 0);
```

**Modification 3**
Add the following signal flow statements to the architecture declaration of the bitsp_processor.

```
bus_read_input block_1 <= dup_bitasu_1_read_io;
dup_bitasu_1_io_outp <= bus_outp16_input block_1 ;
dup_bitasu_1_byte1_valid <= bus_valid_byte1_input block_1(0 downto 0);
dup_bitasu_1_byte2_valid <= bus_valid_byte2_input block_1(0 downto 0);

enable => enablestart,
hold_n1 => hold_bitasu_1,
read_io => dup_bitasu_1_read_io,
io_outp => dup_bitasu_1_io_outp,
byte1_valid => dup_bitasu_1_byte1_valid,
byte2_valid => dup_bitasu_1_byte2_valid,

dup_input block_1_read <= bus_read_input block_1;
bus_outp16_input block_1 <= dup_input block_1_outp16;
bus_valid_byte1_input block_1(0 downto 0) <= dup_input block_1_valid_byte1;
bus_valid_byte2_input block_1(0 downto 0) <= dup_input block_1_valid_byte2;
```

**Modification 4**
Add the component and entity declaration of the input block to the architecture declaration of the bitsp_processor.

```
input block_1 : IO
   port map (
     clk => clk,
     rst_a => rst_a,
     byte => inp_input block_1,
     valid => valid_input block_1,
     eof => eof_input block_1,
     read => dup_input block_1_read,
     req => req_input block_1,
     outp16 => dup_input block_1_outp16,
     valid_byte1 => dup_input block_1_valid_byte1,
     valid_byte2 => dup_input block_1_valid_byte2
   );

component IO
   port (
     clk: in std_logic;
     rst_a: in std_logic;
     byte: in std_logic_vector(7 downto 0);
     valid: in std_logic_vector(0 downto 0);
     eof: in std_logic_vector(0 downto 0);
     read: in std_logic_vector(1 downto 0);
     req: out std_logic_vector(0 downto 0);
     outp16: out std_logic_vector(15 downto 0);
     valid_byte1: out std_logic_vector(0 downto 0);
     valid_byte2: out std_logic_vector(0 downto 0)
   );
```
end component;

**Modification 5**
Add the following ports to the component bitasu declaration in the bitsp_processor architecture declaration.

```
enable: in std_logic;
io_outp: in std_logic_vector(15 downto 0);
byte1_valid: in std_logic_vector(0 downto 0);
byte2_valid: in std_logic_vector(0 downto 0);
hold_n1: out std_logic_vector(0 downto 0);
read_io: out std_logic_vector(1 downto 0);
```

**Modification 6**
Add the VHDL model of the input block to the file that contains the VHDL model for the bitsp_processor.

# Appendix F Changes in SystemC model of bit stream and image processor

This appendix contains a list of changes that should be made to the SystemC generated by A|RT Designer for the bit stream and image processor.

## F.1 *Changes to artd_bitsp.h*

The following changes should be made in the file artd_bitsp.h.

**Modification 1**

Add the following include directive:

```
#ifndef INCLUDED_CTC_INPUT BLOCK"
#include "ctc_input block.h"
#endif
```

**Modification 2**

Add the following lines to the constructor of the class artd_bitsp.

```
Ctc_com_ram *ram_image_data,
Ctc_input block_out *input block_out_1,
```

**Modification 3**

Add the following variables to the list of variables available in the class artd_bitsp.

```
sc_ufixed<1,1> input block_1_req;
sc_ufixed<1,1> input block_1_valid;
sc_ufixed<1,1> input block_1_eof;
sc_ufixed<8,8> input block_1_byte;
sc_ufixed<2,2> bitasu_1_read_io;
sc_ufixed<1,1> bitasu_1_hold;

Ctc_input block *input block_1;
sc_ufixed<16,16> input block_1_outp16;
sc_ufixed<1,1> input block_1_valid_byte1;
sc_ufixed<1,1> input block_1_valid_byte2;

Ctc_input block_out *input block_out_1;
```

## F.2 *Changes to artd_bitsp.cxx*

The following changes should be made in the file artd_bitsp.cxx.

**Modification 1**

Add the following lines to the constructor of the class artd_bitsp.

```
Ctc_com_ram *an_ram_image_data,
Ctc_input block_out *an_input block_out_1,
```

**Modification 2**

Add the following declaration just below the statement `outport_com = an_outport_com` in the artd_bitsp constructor.

```
input block_out_1 = an_input block_out_1;
ram_image_data = an_ram_image_data;
input block_1 = new Ctc_input block("input block_1");
```

**Modification 3**

Insert the following code just before the first `if (!d_hold) statement in the file.`

```
    input block_1->cycle_end(input block_1_req, input block_1_byte, input
                            block_1_valid, input block_1_eof,
                            bitasu_1_read_io, input block_1_outp16, input
                            block_1_valid_byte1, input
                            block_1_valid_byte2);
    input block_out_1->cycle_end(input block_1_req, input block_1_byte,
                            input block_1_valid, input block_1_eof);
```

**Modification 4**

Insert the following code just after the first `if (!d_hold) statement in the file.`

```
    input block_1->next_state(rst);
    input block_out_1->next_state(rst);
```

**Modification 5**

Insert the following code just after the third `switch (d_PC) statement in the file.`

```
    input block_out_1->cycle_start(input block_1_req, input block_1_byte,
                               input block_1_valid, input block_1_eof);
    input block_1->cycle_start(input block_1_req, input block_1_byte, input
                          block_1_valid, input block_1_eof,
                          bitasu_1_read_io, input block_1_outp16, input
                          block_1_valid_byte1, input block_1_valid_byte2);
    bitasu_1_read_io = 0;
```

**Modification 6**

Add the following parameters to the beginning of the cycle cycle_start and cycle end function of the bitasu.

```
    bitasu_1_hold, input block_1_outp16, input block_1_valid_byte1, input
    block_1_valid_byte2, bitasu_1_read_io,
```

## *F.3 Changes to artd_imagep.h*

The following changes should be made in the file artd_imagep.h

**Modification 1**

Add the following line to the constructor of the class artd_imagep.

```
    Ctc_com_ram *ram_image_data,
```

## *F.4 Changes to artd_imagep.cxx*

The following changes should be made in the file artd_imagep.cxx

**Modification 1**

Add the following line to the constructor of the class artd_imagep.

```
    Ctc_com_ram *an_ram_image_data,
```

**Modification 2**

Add the following declaration just below the statement `inport_com = an_inport_com` in the artd_imagep constructor.

```
    ram_image_data = an_ram_image_data;
```