ECE423 Embedded Computer Systems

Lab Manual

by

Rodolfo Pellizzoni and
Muhammad Refaat Soliman

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, 2017

© by the authors

Contents

List of Tables **v**

List of Figures **vii**

I Lab Organization **1**

1 Lab Introduction **2**

1.1 Project Overview and Timeline 4

2 Lab Policies **6**

II Technical Information **8**

3 Preface **9**

4 MJPEG423 Specification **10**

4.1 Sample Code 10

4.2 Color Format 11

4.3 Tables and IDCT Implementation 11

4.4 Lossless Decoding 12

4.5 Optimizing the Code 14

4.6 Frames 14

4.7 File Format 15

4.8 Parallelizing the Decoder 17
4.9 Additional Notes 20

5 Creating the Initial HW Design in Quartus II **21**
5.1 Video IP Cores 34
5.2 Accessing the SD Card 34

6 Performance Estimation **35**
6.1 Application Profiling 35
6.1.1 Performing Meaningful Measurements 36
6.2 Memory Utilization 36
6.3 FPGA Resource Utilization 37
6.4 Memory Size 37

7 HW Accelerators, Streaming Interface and DMA **39**
7.1 Avalon Streaming Interface 40
7.2 Modular Scatter-Gather DMA 41
7.3 Adding a New Component 45
7.4 Cache Coherency 46

8 Multicore Design with Nios II **49**
8.1 Synchronization Primitives 49
8.2 Creating a Multicore Design in Quartus II 50
8.2.1 Multicore Memory Map 50
8.2.2 SOPC Configuration 51
8.2.3 Multicore Software Configurations 52

III Laboratory Projects and Assignment **55**

9 Lab1 **56**
9.1 Activities 56

9.1.1 Complete the HW platform. 56
9.1.2 Execute the sequential MJPEG423 decoder 56
9.1.3 Implement Playback Control with Periodic Timer 57
9.1.4 Profile the Application 59
9.2 Deliverables 59

10 Lab2 **61**
10.1 Activities 61
10.1.1 Create an Hardware Accelerator for IDCT 61
10.1.2 Optional: Create an Hardware Accelerator for Color Combiner . . . 62
10.1.3 Profile the Application and the Platform 62
10.1.4 Optimize Application Period Based on I-P frame Sequencing 63
10.2 Deliverables 63

11 Design Assignment **64**
11.1 Application Scheduling 65
11.1.1 Buffering 65
11.1.2 Synchronization 66
11.1.3 Overheads 67

12 Lab3 **69**
12.1 Activities 69
12.1.1 Create the multicore HW platform 69
12.1.2 Execute the parallel MJPEG423 decoder 69
12.2 Deliverables 70

List of Tables

1.1 Course Timeline 5
1.2 Mark Breakdown (Percentage of Overall Course Grade) 5

9.1 Mark Breakdown For Lab 1 Report 60

10.1 Mark Breakdown For Lab 2 Report 63

12.1 Mark Breakdown For Lab 3 Report 71

List of Figures

1.1 Decoder System Block Diagram 3

4.1 Y'CBCR conversion 11
4.2 2D IDCT Pass 1: Column Processing 13
4.3 2D IDCT Pass 2: Row Processing 13
4.4 I-Frame Processing DAG 18
4.5 P-Frame Processing DAG 19

5.1 Preliminary Qsys System 23
5.2 Video PLL Parameters 24
5.3 Video DMA Parameters 25
5.4 Dual-Clock FIFO Parameters 26
5.5 Video Generator Parameters 27
5.6 Adding Port to LPDDR2 RAM 28
5.7 Qsys Connections of Lab 1 29
5.8 Qsys Top Level Connections 31
5.9 Adding Timestamp Timer in BSP Editor 32
5.10 Linker Script Tab in BSP Editor 33

6.1 Extracting Program Size from Build Summary 38

7.1 Basic Accelerator Design Block Diagram 40
7.2 Interface Transfer Diagram, READY_LATENCY = 0 41
7.3 Memory Controller Port Configuration 42

7.4 DMA Core Configuration, Avalon-ST Interface to Memory 43
7.5 Component Editor for IDCT Accelerator 47

8.1 Qsys Connections, Two Cores 53

11.1 Example Synchronization 67

Part I

Lab Organization

Chapter 1

Lab Introduction

Welcome to ECE423. This chapter will introduce you to our lab project and discuss the organization of the lab manual.

The overall goal of the ECE423 lab is to design and implement a video decoder on an Altera Cyclone V FPGA board for the MJPEG423 format. As you can surmise from the name, the MJPEG423 format has been created specifically for this lab. It implements a subset of the functionalities in typical full-fledged video codecs such as MPEG; it is simple enough to be implementable as part of a course project, while retaining the major characteristics (and complexities) of the domain. The project builds on top of what you learned in the ECE224/325 lab, and in particular lab 2, where you implemented a sound decoder, but introduces new challenges:

- Parallelism: the MJPEG423 decoder is much more computationally intensive than the sound decoder of ECE224. Hence, you will not be able to run it at a satisfactory speed on a single processor core (Nios II). Therefore, you will need to run the application in parallel on multiple Processing Elements (PE).
- HW accelerators: some of the functional blocks in the application can be more efficiently realized as custom hw components in VHDL rather than running as software modules on a general purpose Nios II core. Hence, you will design one or more hw accelerators that can be run together with the software parts of the applications executed on the core(s). Note that your job is mainly to understand how to integrate these hw blocks with the Nios II - we will not ask you to code complex VHDL modules like in ECE327.
- HW/SW co-design: as you proceed in the labs, you will notice that you have a lot of choice in the implementation of the decoder - whether you implement each functionality in hw or sw, how many cores and hw coprocessors you use, how to

schedule the execution of the various functional blocks, etc. As a matter of fact, a significant portion of the lecture time will be spent discussing how to properly co-design the hw/sw components in a system-on-chip design. The lab gives you a way to concretely practice what is taught in class.

Figure 1.1: Decoder System Block Diagram

A high-level block diagram of the specified system is provided in Figure 1.1. A MJPEG423 video file is provided on Secure Digital card (SD). The video file encodes a sequence of compressed frames (images), that you must show to the user at a specified and constant frequency (frame rate). Using a provided SD card hardware interface, you read compressed frame information from the file and store it in main memory. Then the decoder application (running on one or more Nios II with or without custom hw coprocessors) decodes each frame into a bitmap, i.e., a representation of the frame where each pixel in the image is associated with a color in Red-Green-Blue (RGB) format. The obtained frame bitmap is then output to the monitor using a set of provided VGA IP blocks. Your objective is to ensure that you can meet frame rate requirements of the application while minimizing the consumption of FPGA platform resources; or assuming you cannot meet the frame rate requirement, simply maximize the frame rate, potentially consuming all available FPGA resources.

The rest of the manual is organized as follows. Chapter 2 discusses how to form groups, deadlines, and other relevant lab policies. Part II details all technical issues relevant to completing the project. In particular, Chapter 4 provides the specification of the MJPEG423 format, while Chapters 5-8 discuss essential implementation concerns that you likely did not encounter before. Finally, Part III details the actual lab deliverables. A overview of the activities in each lab and a timeline for the completion of the project are further discussed in the next section.

1.1 Project Overview and Timeline

There are six scheduled lab sessions. The lab sessions are held on even weeks. Some of the class tutorials will also be used to introduce the various lab parts. The course project workload consists of three Labs, covering two lab sessions each (Lab1a and Lab1b, Lab2a and Lab2b, Lab3a and Lab3b), and a Design Assignment. Each lab builds on top of the previous one. The Design Assignment complements the activities in the lab; it has no deliverable since the output of this stage is included in the final project report, but the teaching team will meet with each group in the week of Lab2b to ensure that the selected design is implementable in Lab 3.

In Lab1, you create a basic single-core SoC system to execute the sequential MJPEG423 decoder. We provide the code for the decoder, but to execute it on the FPGA board, you will need to interface it with the SD and VGA libraries (which are similarly already provided). Then, you will profile the system to understand how long each functional block in the application takes to execute. This will give you an idea of which areas of the application must be optimized.

In Lab2, you will create specific a hw coprocessor to speed-up the execution of the most critical part of the application - the inverse discrete cosine transform (IDCT) . You will also profile the performance of the hw components and gather metrics on FPGA resource utilization.

In the Design Assignment, you will use the performance characterization derived in Lab1 and Lab2 to produce an actual design for the system, that is, specify: 1. which PEs you will use, their number and configuration; 2. the mapping of functional (software) blocks to PEs. Finally, in Lab3 you will execute the parallelized application based on the design you created in the Design Assignment.

The required lab deliverables are described in details in Part III. Table 1.1 summarizes the deadlines for all deliverables. The mark breakdown for the course project is provided in Table 1.2. The teaching team reserves the right to assign bonus points to rewards groups that are able to devise particularly clever solutions/optimizations or go beyond the stated lab requirements.

All course project activities are to be performed in teams of 3 students. Details on how to form groups are provided in Chapter 2. Note that since the course workload is significant, it is essential that you clearly organize and manage responsibilities within the team. Being 4th year students, you are free to manage your team as you see fit, but here are a few suggestions:

- Elect a project leader to manage assignments within the team and keep people working on time.

- Play to your strengths: many teams are likely to include a mix of computer engineering, electrical engineering and mechatronics students. Some of you might have co-oped in hardware companies, some in software companies. Assign work items based on your individual strengths and expertises.

- Ensure that all main project areas are covered: you might want to assign responsibilities within the team based on key project technical areas. As you will figure out, there are three main areas to be covered for a successful project: 1. application (understanding how the mjpeg423 decoder works); 2. hardware (how to create a FPGA platform in Quartus II and how to interface hw accelerators); 3. integration (how to put hw and sw together, profile, run and synchronize the application).

Submit lab groups	Monday Jan 9 8:30AM
Lab1a session	Tuesday Jan 10 7:00PM-10:00PM
Lab1b session and demo	Tuesday Jan 24 7:00PM-10:00PM
Lab2a session	Tuesday Feb 7 7:00PM-10:00PM
Lab2b session and demo	Tuesday Feb 28 7:00PM-10:00PM
Lab3a session	Tuesday Mar 14 7:00PM-10:00PM
Lab3b session and demo	Tuesday Mar 28 7:00PM-10:00PM
Lab 1 mini-report due	Monday Jan 30 8:30AM
Lab 2 mini-report due	Monday Mar 6 8:30AM
Final report due	Monday Apr 3 11:59PM

Table 1.1: Course Timeline

Total Lab	35%
Lab1 demo	5%
Lab1 report	4%
Lab2 demo	5%
Lab2 report	4%
Lab3 demo	4%
Lab3 performance	4%
Final Report	9%

Table 1.2: Mark Breakdown (Percentage of Overall Course Grade)

Chapter 2

Lab Policies

Lab Groups All lab activities are performed in groups of 3 students. You are free to create the groups, but it is strongly advised that each group satisfies the following requirement: at least one student in the group should have experience with digital design in VHDL/Verilog. Note that attending or having passed ECE327 satisfies the requirement (in fact, the ECE327 project is significantly more complex from a digital design perspective). The rule is in place to ensure that you do not run into issues in Lab 2. It is also advisable that at least one student in each group is knowledgeable about synchronization, as taught in OS courses (ECE254 / 354 / MTE 241 or equivalent).

A sufficient number of groups for all students in the course have been created in Learn. You can form your group by simply joining one of the groups on Learn. **Make sure you form your group by Monday Jan 9 at 8:30AM.** To facilitate forming groups, we have created a discussion forum on learn. If you cannot find a group by the stated deadline, please send an email to the Lab Instructor and CC to the Course Instructor.

Note 1: the teaching team reserves the right to rearrange the groups should any exceptional situation occur.

Note 2: if the number of students in the course is not divisible by 3, we will form a suitable number of groups with 4 students.

Lab Access Lab sessions are scheduled on Tuesday 7:00PM - 10:00PM in E2 2364 every other week. The labs are equipped with the required software, boards, cables and SD cards. After-hour access to the lab will be available this term; the access code will be published on Learn.

Deliverables Each lab's deliverables consist of a demo and a report submission. Lab demos will be conducted during the lab sessions 1b/2b/3b. The report must be submitted

through Learn; there are separate dropboxes for each. In addition, you are required to submit all code used in each demo immediately after the demo itself.

Grace Days There are four grace days that can be used for late submissions of the lab reports for lab 1 and 2. When you use up all your grace days, a 10% per day late penalty will be applied to a late submission. Please be advised that to simplify the book-keeping, late submission is counted in a unit of day rather than hour or minute. An hour late submission is one day late, so does a fifteen hour late submission. Grace days cannot be used for the Final Report, since it is due on the last day of class.

Note on Plagiarism You are allowed to discuss general solutions to the lab assignments with other groups, but you cannot share code outside your lab group. Do not upload any portion of the lab material (including documents, code, libraries, or any deliverable) to a publicly accessible website, including a public repository. You can use a private repository to version control your code if you so desire, as long as you have proper access control and no anonymous access is allowed.

Part II

Technical Information

Chapter 3

Preface

This part of the lab manual contains all the technical information that is required to complete the lab projects specified in Part III and implement a MJPEG423 decoder on an Altera DE2 board.

Note that the present manual assumes familiarity with the content of the ECE224 / 325 lab project. In particular, we assume that you should be able to create a custom microprocessor design using Quartus. For your reference, the ECE224 lab manual and other required files are available on the ECE423 learn website.

Chapter 5 discusses how to create the initial HW design in Quartus II that will be used in Lab1. The design is very similar to the one in 224, but you will need to add the SD interface as well as VGA IP blocks to output to the monitor. Chapter 6 discusses how to profile the application (needed for Lab1 and Lab2) and gather resource utilization metrics (needed in Lab2). Chapter 7 covers how to create HW accelerators and interface them to the rest of the system using the existing DMA controller IP, which is the main topic of Lab2. Finally, Chapter 8 explains how to create a multicore Nios II design in Quartus II, as well as important considerations on how to code on such systems.

Chapter 4

MJPEG423 Specification

This chapter provides the specification of the MJPEG423 standard. Since the standard is heavily based on (a simplified version of) the JPEG baseline decoder, you should start by getting familiar with the JPEG standard. The original description of the standard can be found on the learn website as jpeg_wallace.pdf. We also provide the latest complete specification in jpeg-spec.pdf, albeit you will not likely need to read this document. MJPEG423 employs a variation of the baseline JPEG decoder with 8 bit color components and Y'CBCR color format. After reading through jpeg_wallace.pdf, you should be familiar with: 1. the key processing steps of entropy decoding, dequantization and IDCT in Section 4; 2. the color components in Section 6.1; 3. the operation of the baseline decoder in Section 7. The remaining sections in this chapter will detail the MJPEG423 file format and the differences between the MJPEG423 decoder and the JPEG baseline.

4.1 Sample Code

Sample code for the MJPEG423 decoder is available on learn. The application is provided as a collection of C header and source files, using the C99 standard. The application takes as input a MJPEG423 file and produces as output a series of bmp files, one for each frame in the video. Note that the application executes sequentially as fast as possible, i.e., it makes no attempt to match the specified frame rate of the video.

It is suggested that you look through the code of the sample decoder as you read this chapter. Note that each key functional block (lossless decoding, IDCT, color conversion) has been isolated in a different function and source file for simplicity. Finally, the open-source lib bmp library is used to save the reconstructed image data to a bmp.

Apart for the sample MJPEG423 decoder, on learn we also provide a set of example

MJPEG423 video files, plus a MJPEG423 encoder application that can be used to convert a set of frame bitmaps into a MJPEG423 video file.

4.2 Color Format

When stored in a bitmap or output to a screen, images are most commonly encoded in the RGB color format, where the color of each pixel is represented by the intensity of the red, green and blue color components (for the rest of this discussion, we assume that the intensity of each component is represented with an 8 bit value, i.e., between 0 and 255). However, the RGB color format is unsuitable for compression, since it does not accurately represent the way that the human eye perceives differences in color. Therefore, the MJPEG423 format stores pixel color in the alternative Y'CBCR format, composed of a brightness component (Y') and two chrominance components (CB, blue chrominance, and CR, red chrominance). The equations in Figure 4.1 allow you to transform from the RGB color space to the Y'CBCR color space.

$$
\begin{aligned}
Y' &= 0 + (0.299 \cdot R'_D) + (0.587 \cdot G'_D) + (0.114 \cdot B'_D) \\
C_B &= 128 - (0.168736 \cdot R'_D) - (0.331264 \cdot G'_D) + (0.5 \cdot B'_D) \\
C_R &= 128 + (0.5 \cdot R'_D) - (0.418688 \cdot G'_D) - (0.081312 \cdot B'_D)
\end{aligned}
$$

Figure 4.1: Y'CBCR conversion

Note that the equations in Figure 4.1 are expressed in terms of floating point multiplications. However, the Nios II does not have a floating point unit. Hence, the sample decoder implements the conversion using fixed-point arithmetic on 24bits in the *ycbcr_to_rgb* function (read the function code for details); note that the function accepts as inputs one Y' component 8x8 blocks, one CB component 8x8 block and one CR component 8x8 block and outputs 8 lines of 8 pixels in RGB format.

4.3 Tables and IDCT Implementation

MJPEG423 uses fixed quantization tables; the tables are hardcoded in the specification and provided in the sample application as file *tables.c*. There are different tables for the Y' component and for the CBCR components.

As specified in the JPEG standard, the precision of the DCT transforms depends strictly on the implementation. Again for performance reasons, MJPEG423 uses a fixed-point computation. The chosen implementation is based on the method by Loeffler et al.; for

reference we provide the corresponding paper as idct1D.pdf on Learn, but you do not need to understand the implementation details past what is described in this section.

A 2-dimensional IDCT can be implemented in terms of a 1-dimensional IDCT by following the approach illustrated in Figures 4.2, 4.3: first, the 1D IDCT is applied to each of the 8 columns of the input component matrix (DCAC), and the result is stored in a temporary workspace matrix. Then, the same 1D IDCT transformation is applied to each row in the workspace matrix, resulting in the final row of the output color component matrix (block); this are called Pass 1 and Pass 2 in idct.c, respectively. There are, however, a few peculiarities to be aware of:

- The values computed by Pass 1 and by Pass 2 are scaled by a factor of $2\sqrt{2}$ compared to the true 1D IDCT values. This means that the output values are actually scaled by a factor of 8. Hence, at end of Pass 2, the outputs are left-shifted by 3 bits to obtain the correct values.
- To perform fixed-point computations with sufficient precision, in each pass the inputs are first multiplied by a factor $2^{\text{CONST_BITS}}$ (where CONST_BITS = 13), and then divided by the same factor at the end of the pass.
- Furthermore, to increase the precision of the computation, the temporary values stored in workspace are further multiplied by a factor $2^{\text{PASS1_BITS}} = 4$; hence, to obtain the actual values at the end of Pass 2, we need to divide them by the same factor.

In summary: at the end of Pass 1 (lines 107-114), the resulting values are left shifted by CONST_BITS − PASS1_BITS bits to account for the fixed-point scaling in Pass 1 and the PASS1_BITS scaling; while at the end of Pass 2 (lines 179-186), the final values are left shifted by CONST_BITS + PASS1_BITS + 3 to account for the fixed-point scaling in Pass 2, the PASS1_BITS scaling, and the 1D IDCT scaling by 8. At the end of Pass 2 we also normalize the results to ensure that they fall in the range $[0, 255]$ despite any possible numeric error. Apart from the different descaling and the normalization at the end of the pass, the operations performed by Pass 1 and Pass 2 are exactly the same (you will use this property in Lab 2 when you implement a hw coprocessor for the 2D IDCT based on provided 1D IDCT VHDL code).

4.4 Lossless Decoding

The "entropy coding" step described in the JPEG standard is really a mix of various *lossless coding* techniques. The decoding is implemented in the lossless_decode.c file. MJPEG423

Figure 4.2: 2D IDCT Pass 1: Column Processing

Figure 4.3: 2D IDCT Pass 2: Row Processing

uses a scheme very similar to the baseline JPEG: Variable Length Integer (VLI) coding is used to encode each DC or AC component into a SIZE and an AMPLITUDE; each DC component is the then encoded as (SIZE, AMPLITUDE), while AC components are encoded as (RUNLENGTH, SIZE, AMPLITUDE). The same EOB (end of block) and ZRL (runlength ≥ 16) codes are used, together with the zig-zag sequence of Figure 3 in jpeg_wallace.pdf. The only difference is that rather than further encoding RUNLENGTH and SIZE using Huffman encoding, they are for simplicity represented with 4 bits each. Note that since the maximum RUNLENGTH is 15 and the maximum SIZE is 11, 4 bits are sufficient.

Finally, notice that the lossless_decode function performs **both lossless decoding and dequantization**. This is an obvious optimization - any coefficient equal to zero does not need to be dequantized.

4.5 Optimizing the Code

Note that the precision and quality of the decoded image is strictly dependent on the algorithms used to perform the IDCT and color conversion steps. While you are allowed to modify (and potentially optimize) the code of the decoder, you cannot change the fixed-point precision used in the idct and ycbcr_to_rgb functions or change the employed IDCT algorithm (there exist other algorithms that trade off precision for additional speed of execution).

4.6 Frames

The objective of the JPEG standard is to encode single images. On the other hand, the objective of MJPEG423 is to encode a video, i.e., a sequence of images (frames) all of the same size, that must be played back at a constant frequency (frame rate). In many cases, successive frames (or significant portions of successive frames) can be very similar to each other in terms of pixel color. Hence, we can further optimize the compression ratio by taking advantage of such similarity. To this end, MJPEG423 defines two types of frames: I(index) frames and P(regressive) frames.

- **I-frames:** I-frames are stored just like normal JPEG images. Each of the 63 AC coefficients in a block is stored in absolute value, while the DC coefficients use differencing (let DC_i be the DC component value obtained after lossless decoding for block i; then to obtain the real DC component DC'_i to send to IDCT, you must compute $DC'_0 = DC_0$ for the top-left block 0 and $DC'_i = DC'_{i-1} + DC_i$ for every other block).

- **P-frames:** P-frames use differential encoding; this means that both the AC and DC coefficients are stored as the difference between successive images. More in detail, let $AC_i^j(x, y)$ be the (x, y) AC coefficient value for block i of frame j obtained after lossless decoding; then the real AC component $AC_i'^j(x, y)$ for frame j can be obtained as $AC_i'^j(x, y) = AC_i'^{j-1}(x, y) + AC_i^j(x, y)$ (similarly, DC differencing is not used for P-frames; instead, the real component is obtained as $DC_i'^j = DC_i'^{j-1} + DC_i^j$).

The first frame in a video is always an I-frame; successive frames can either be P-frames or I-frames. Note that decoding a P-frame relies on the sequence of previous frames since the last I-frame in the stream; this poses a problem if a user watching a video wants to jump backwards/forward in the stream, since to display a given P-frame we need to decode all frames since the previous I-frame. For this reason, the maximum number of P-frames between successive I-frames should be bounded. The sample MJPEG423 encoder inserts an I-frame whenever the amount of file space saved by encoding the frame as a P-frame instead is not significant, or after a maximum configurable number of frames have passed since the last I-frame was inserted.

4.7 File Format

Each MJPEG423 file is composed of three consecutive sections: a header section, a payload, and a trailer. The header and trailer contain control information. The payload contains the sequence of frames stored in the video. In what follows, uint32 represents a 32-bits unsigned integer. Value are stored in little-endian format[1].

Header The header is always composed of 6 4-bytes fields, as follows:

1. uint32 num_frames: the total number of frames in the video.
2. uint32 num_iframes: the total number of i-frames in the video.
3. uint32 x_size: the width of the video in number of pixels. This value must be a multiple of 16.
4. uint32 y_size: the height of the video in number of pixels. This value must be a multiple of 16.
5. uint32 payload_size: the total size of the payload in number of bytes. Since frames are aligned to 32 bits, this value is always a multiple of 4.

[1]Note that all functional blocks in the decoder are written to be endianess-independent; however, the main decode and encode functions use fread and fwrite to output uint32 values, hence they can only be executed on little-endian machines. Luckily, both Intel machines and the Nios II are little-endian.

Payload The payload is composed by a sequence of frames, one after the other. Each frame is aligned to 32 bits (4 bytes), and composed of the following fields:

1. uint32 frame_size: the total size of the frame in number of bytes. Always a multiple of 4.
2. uint32 type: 0 for an I-frame, 1 for a P-frame.
3. uint32 Ysize: the size of the bitstream for the Y' component in number of bytes.
4. uint32 CBsize: the size of the bitstream for the CB component in number of bytes.
5. Ybitstream: the bitstream for the Y' component produced by Huffman encoding in the encoder. Note that if the frame starts at byte index k in the file, the Ybitstream starts at byte index $k+16$ (since the previous 4 fields take 16 bytes total). Also note that there is no requirement that the bitstream occupies an integer number of bytes; in this case, the remaining bits in the last byte of the bitstream are used as padding and set to 0 by the encoder; this ensures that the successive CBbitstream starts at the beginning of a byte (i.e., byte aligned).
6. CBbitstream: as above, but for the CB_bitstream. Note that if the frame starts at byte index k in the file, the CB_bitstream starts at byte index $k + 16 +$ Ysize.
7. CRbitstream: as above, but for the CR_bitstream. Note that if the frame starts at byte index k in the file, the CR_bitstream starts at byte index $k + 16 +$ Ysize $+$ CBsize.

Trailer The trailer contains information about I-frames. The total length of the trailer in bytes is $8\cdot$ num_iframe. For each I-frame, the following information is provided:

1. uint32 frame_index: the index of the frame (i.e., each frame is indexed starting from frame 0 for the first frame in the video).
2. uint32 frame_position: the byte index from the beginning of the file at which the frame information starts.

The trailer information is used to allow a user to jump back and forth in the video. As an example, assume that the trailer contains information for two successive I-frames with indexes 1204 and 1267, and the user attempts to jump to frame 1220. Using the trailer information, the decoder can figure out the I-frame preceding P-frame 1220 is frame 1204, and then restart decoding from that frame.

4.8 Parallelizing the Decoder

The provided sample decoder application is entirely sequential. As you will quickly figure out during Lab 1, while the sample decoder runs reasonably fast on a modern PC processor, it is way too slow to display videos at acceptable frame rates when executed on a single Nios II (by roughly one order of magnitude). Hence, your goal by the end of Lab 3 will be to implement a more efficient parallel decoder.

To decide how to parallelize the application, it is first necessary to understand the *data-level parallelism* in the application, i.e., which functional blocks can be executed in parallel on which data. Figures 4.4, 4.5 show the execution of the decoder for either a I or P-frame of 16x8 pixels (two 8x8 blocks), where the decoder is modeled as a Directed Acyclic Graph (DAG). In the figure, nodes (circles) represent functional blocks, and are denoted with the name of the corresponding function in the sample application. Blocks (squares) instead represents data elements, typically part of an array, and are denoted with the name of the corresponding variable in the sample application. An arrow from a data block to a node means that the node must read the content of that data block, while an arrow from a node to a data block means that the node changes the content of that data block; hence, arrows represent precedence constraints, in the sense that if a node A writes to a data block and the same data block is read by a node B, then node A must finish executing before node B can be executed. Similarly, an arrow between two nodes represent a direct precedence constraint between the two nodes (due to local variables). However, nodes without precedence constraints can be run in parallel.

Note that lossless_decode can be applied in parallel to each of three component bitstreams (for Y', CB and CR). lossless_decode produces a set of 8x8 blocks of dequantized coefficients for each component, encoded in the YDCAC, CBDCAC and CRDCAC matrixes. In the case of an I-frame, lossless_decode only needs as input the corresponding bitstream; in the case of a P-frame, it also needs the YDAC, CBDCAC and CRDCAC matrixes computed at the previous frame. Each IDCT operation then takes a 8x8 block of coefficients and produces an 8x8 pixel color block for that color component, stored in either the Yblock, CBblock or CRblock matrix. Finally, the ycbcr_to_rgb function takes one 8x8 color block of each component and outputs 8 lines of 8 RGB bits in the 16x8 rgbblock matrix (i.e., an 8x8 block in see matrix; see also Section 4.2).

When you produce a design for the parallel decoder in the Design Assignment, you will need to perform mapping, i.e., decide how to assign the various functional blocks to Processing Elements (PEs - Nios II and HW accelerators). Each node in the DAG (i.e, an instance of a functional block executed on one image block) should execute on one PE, but apart from this limitation, you are free to explore different mappings. In particular, different nodes of the same functional blocks could be executed either on the same PE or on different PE (ex: you could split the lossless_decode nodes on 2 Nios II cores). This

Figure 4.4: I-Frame Processing DAG

Figure 4.5: P-Frame Processing DAG

said, based on the data-level parallelism of the application, we can make a set of important observations:

1. You want to execute all instances of lossless_decode for the same color component on one PE, since there are precedence constraints among them. On the other hand, you could execute lossless_decode for the Y' component on one PE, for the CB component on a second one, and for the CR component on a third one. This said, take into account the following: 1. lossless_decode takes much longer on I-frames than P-frames, since P-frames are more heavily compressed; 2. lossless_decode on Y' component takes longer than on CB/CR components, since the chrominance components are more heavily quantized than the luminance component. For this reason, when you profile lossless_decode in Lab 1, you should treat I and P-frames and Y' and CB/CR components differently.

2. On the other hand, the IDCT and ycbcr_to_rgb operations can be highly parallelized; in this sense, your performance is likely to be bounded by the amount of available processing and memory resources on the board.

3. Finally, notice that each block can be processed in a pipeline. In other words, you do not need to wait for each of the first two stages (lossless_decode and IDCT) to finish processing all its blocks before you move to the next stage. Instead, immediately after lossless_decode outputs the first 8x8 block for a color component, you can immediately start computing the IDCT on that block; and after you computed the IDCT for the first blocks of each component, you can immediately make the first call to ycbcr_to_rgb.

4.9 Additional Notes

Since there are various files shared between the encoder and decoder sample applications, the two are provided in the same source code archive. The main file shows how to call either the encoder or decoder. Each application is provided in a separate directory, while the common directory contains shared files. When you port the decoder to the Nios II platform in Lab1, you will only need the files in the decoder and common directories.

The common directory contains two important header files, mjpeg423_types.h and util.h. The first file contains essential definitions of types and data structures used throughout the application (including the matrixes in Figures 4.4, 4.5). The second file contains a set of utility and debug flags. You should probably start by studying these two files before you take a look at mjpeg423_decoder.c, the main decoder application file. All the code have been thoroughly commented.

Chapter 5

Creating the Initial HW Design in Quartus II

This chapter discusses how to create the initial hw design under Quartus II and Qsys that will be used in Lab1 and serve as the basis for Lab2 and Lab3. As discussed in Chapter 1, we assume familiarity with the Nios II tutorial run in ECE224, and in general with the Altera development tools. If you need a refresh, the lab manual for ECE224 is available on learn. **Avoid compiling the project from your N: drive**, as this will break the tools.

Start by downloading the ece423_prefab.zip file from Learn. The file contains a working, pre-made Quartus II 15.1 project for the Cyclone V board. The project already contains a Qsys module implementing a custom Nios II system. The following peripherals have been added to the Qsys module:

- Clocks (clk_125, clk_50) running at 125 Mhz and 50 Mhz respectively.
- Nios II/f processor (cpu).
- System ID (sysid).
- JTAG UART (jtag_uart).
- LPDDR2 RAM Controller (lpddr2), connected to a 512MB external SDRAM memory module.
- SRAM Controller (sram, sram_sharer, sram_bridge), connected to a 512KB external SRAM memory module.
- Interval Timers (timer_0, timer_1).

- PIO for the Push Buttons (key) configured as: 4 bits inputs, synchronous capture on any edge, and generate edge interrupt.
- PIO for the red LEDs (ledr) and green LEDs (ledg). While LED are not strictly required for the project, they can help with debugging.
- I2C PIOs (i2c_scl, i2c_sda) to be used to configure the HDMI chip on the board.

Figure 5.1 shows the provided Qsys system.

To complete the hardware design, you will add the components required to handle the video output and the SD interface to the provided project. In this part we focus on explaining the required steps to obtain a design compatible with the requirements of Lab1; an explanation of how the video IP cores work is provided in Section 5.1. Start by opening Qsys on the provided project, then add the following components (to find a component, use the search box in IP Catalog on the left of Qsys window. Also, to export a port, refer to the Export column for that port and double-click to export):

- Add the *SD controller* (sd_cont). Connect the avalon slave port (slave) of (sd_cont) to the (data_master) of (cpu) and the avalon master port (master) of (sd_cont) to (avl_0) of (lpddr2). Connect both clocks (clock, sd_clk) to the 50 Mhz clock (clk_50). Finally, export the (sd) conduit with the name (sd).
- The video output needs a 25 Mhz clock. Add *Altera PLL* (video_pll) and modify the parameters as shown in Figure 5.2. Connect the reference clock (refclk) to (clk_50). We need to export this clock to the top level design to be connected to the HDMI clock, so add a *Clock Bridge* (video_clk) and connect its reference clock (in_clock) to the output of the PLL and export its output (out_clock) with the name (video_clk).
- Next add *Modular Scatter-Gather DMA* (video_dma). Configure the DMA parameters as indicated in Figure 5.3. The control ports of the DMA (csr, descriptor_slave) are connected to the (data_master) port of (cpu) and the interrupt port (csr_irq) is connected to the interrupt port (irq) of (cpu). The DMA clock (clk) is connected to (clk_125). We need to connect the DMA read port (mm_read) to the LPDDR2 RAM, so add a read-only port to the LPDDR2 RAM as depicted in Figure 5.6. Connect (mm_read) of (video_dma) to new memory port (avl_1). The new port clocks (mp_cmd_clk_1) and (mp_rfifo_clk_1) should be connected to (clk_125).
- Add *Avalon FIFO memory* (video_fifo) and configure it as shown in Figure 5.4. Connect the streaming output (st_source) of (video_dma) to the input (in) of (video_fifo). The FIFO has two clocks. The input clock (clk_in) is (clk_125) which is the same as the clock of (video_dma). The output clock (clk_out) is the video clock which is the output of (video_pll).

Figure 5.1: Preliminary Qsys System

Figure 5.2: Video PLL Parameters

Figure 5.3: Video DMA Parameters

Figure 5.4: Dual-Clock FIFO Parameters

Figure 5.5: Video Generator Parameters

Figure 5.6: Adding Port to LPDDR2 RAM

Figure 5.7: Qsys Connections of Lab 1

- Add *Pixel Converter* (pixel_conv) and choose the Source symbols per beat to be 1. Connect the output (out) of (video_fifo) to the input (in) of (pixel_conv). The clock of (pixel_conv) is the video clock, so connect it to the output of (video_pll).

- Finally, add *Video Sync Generator* (video) component. Configure it using the parameters in Figure 5.5 to support a video resolution of 640*480. The clock (clk) is the video clock and the input (in) is connected to the output (out) of (pixel_conv). The video signals are exported to the top level from the (sync) conduit with the name (video).

- Follow Figure 5.7 to complete any missing connections in your design. If you have followed the description so far, the only missing connections should be the reset ports of all added components, which must be connected to (reset_out) port of (reset_controller_0).

- Make sure to Auto-Assign Base Address from System menu to eliminate any conflicts in the assigned base addresses.

At this point, you should have no remaining error messages in the information tab at the bottom of Qsys (info messages are fine). Figure 5.7 shows the resulting configuration (yours might have a different order or different addresses, which is fine). You can now re-generate the Qsys system.

After you finish generating your Qsys system, you need to connect your new components to the top level design. Go back to Quartus and double click on the top level design (ECE423_C5G) to open the HDL design file. Scroll down to the Qsys component (ECE423_QSYS) and add the new connections as shown in Figure 5.8, then compile the project.

Once you finish compiling the hardware design, you can create a new software project in the Nios II Software Build Tool. Make sure to create an application project + BSP project using the Altera HAL API. Note that for the purpose of profiling, we use one of the timers (timer_0) as a timestamp timer. To set up the timestamp timer, right-click on the BSP project and choose NIOS II → BSP Editor. Under the Main tab, specify the timestamp_timer to be (timer_0) as shown in Figure 5.9. You can use the default configuration for timer_1 - you will use it as a periodic timer to ensure that video frames are sent to the monitor at a constant frame rate. Also note that we use two memories (SRAM, LPDDR2). SRAM is used for the code while LPDDR2 is used for the data. In the BSP Editor, go to Linker Script tab and make sure that .text section is located in (sram) and all the data sections are in (lpddr2) as in Figure 5.10. To use the SD card and video API, download ece423_lib.zip from learn and extract it in your project folder.

```
ECE423_QSYS u0 (
  .clk_125_clk                                  (CLOCK_125_p),
  .clk_50_clk                                   (CLOCK_50_B7A),
  .reset_reset_n                                (CPU_RESET_n),

  .lpddr2_mem_ca                                (DDR2LP_CA),
  .lpddr2_mem_ck                                (DDR2LP_CK_p),
  .lpddr2_mem_ck_n                              (DDR2LP_CK_n),
  .lpddr2_mem_cke                               (DDR2LP_CKE),
  .lpddr2_mem_cs_n                              (DDR2LP_CS_n),
  .lpddr2_mem_dm                                (DDR2LP_DM),
  .lpddr2_mem_dq                                (DDR2LP_DQ),
  .lpddr2_mem_dqs                               (DDR2LP_DQS_p),
  .lpddr2_mem_dqs_n                             (DDR2LP_DQS_n),
  .lpddr2_oct_rzqin                             (DDR2LP_OCT_RZQ),

  .lpddr2_pll_ref_clk_clk                       (CLOCK_50_B5B),
  .lpddr2_status_local_init_done                (lpddr2_local_init_done),
  .lpddr2_status_local_cal_success              (lpddr2_local_cal_success),
  .lpddr2_status_local_cal_fail                 (lpddr2_local_cal_fail),
  .lpddr2_global_reset_reset_n                  (1'b1),

  .key_export                                   (KEY),
  .ledr_export                                  (LEDR[7:0]),
  .ledg_export                                  (LEDG[7:0]),

  .sram_bridge_out_sram_tcm_data_out            (SRAM_D),
  .sram_bridge_out_sram_tcm_address_out         ({SRAM_A,sram_addr_dummy}),
  .sram_bridge_out_sram_tcm_outputenable_n_out  (SRAM_OE_n),
  .sram_bridge_out_sram_tcm_chipselect_n_out    (SRAM_CE_n),
  .sram_bridge_out_sram_tcm_write_n_out         (SRAM_WE_n),
  .sram_bridge_out_sram_tcm_byteenable_n_out    ({SRAM_UB_n,SRAM_LB_n}),

  .i2c_sda_export                               (I2C_SDA),
  .i2c_scl_export                               (I2C_SCL),

  .video_clk_clk                                (HDMI_TX_CLK),
  .video_RGB_OUT                                (HDMI_TX_D),
  .video_HD                                     (HDMI_TX_HS),
  .video_VD                                     (HDMI_TX_VS),
  .video_DEN                                    (HDMI_TX_DE),

  .sd_sd_clk                                    (SD_CLK),
  .sd_sd_cmd                                    (SD_CMD),
  .sd_sd_dat                                    (SD_DAT)
 );
```

Figure 5.8: Qsys Top Level Connections

Figure 5.9: Adding Timestamp Timer in BSP Editor

Figure 5.10: Linker Script Tab in BSP Editor

5.1 Video IP Cores

The video sync generator (video) generates the video signal that flows to the HDMI chip on the board. So, it continuously outputs to the monitor, drawing a complete frame 60 times per second (which is higher than the typical frame rate of 24fps used by videos). Since we uses a resolution of 640x480 pixels and requires 24 color bits per pixel, it effectively needs to be fed 640x480x24x60bits per second. Since this is quite a bit of data, instead of doing so in software we use a DMA controller. 60 times per second, the Modular Scatter-Gather DMA (video_dma) takes the frame information from a specified location in main memory (frame buffer) and transfers them to the video generator. The DMA is configured by the processor using the video API explained in the document ECE423_API_REF.pdf on learn.

The video DMA (video_dma) is clocked using the 125 Mhz clock while the video output is clocked using 25 Mhz. Hence, we use a dual-clock FIFO (video_fifo) to regulate the flow of pixels between them. The DMA reads one word (4 bytes) while the video output uses 3 bytes for each pixel; this ensures that each pixel is word-aligned in the frame buffer. The pixel converter (pixel_conv) is then used to extract the 3-byte pixel from the 4 bytes read by the video DMA. More in details, the video uses the Y'CBCR color format, where the 32 color bits (4 bytes) read by the video DMA for each pixel are arranged as follows: bits 7-0 are the CB component; bits 15-8 are the Y' component; bits 23-16 are the CR components; and bits 31-24 are don't care (dropped by the pixel converter).

A final important note is relative to using the frame buffer. Since the video generator draws one line at the time, modifying the content of the frame buffer to show the next video frame while the video generator is outputting a frame to the monitor is problematic - it might result in the top part of the monitor displaying the content of a frame, and the bottom of the monitor displaying the content of the next frame. This effect, known as tearing, is visually jarring and should be avoided. The solution is to let the video generator operate on one frame buffer, while the application updates a different buffer. Then after the video generator finishes drawing the current frame, it displays the next buffer. The number of buffers can be specified using the video API.

5.2 Accessing the SD Card

We use an SD card controller that exploits the 4-bit SD bus to maximize the transfer rate. The SD controller has an integrated DMA that directly transfer the required data to the memory without the need to copy one sector at a time using the processor. We provide a FAT file system library that controls the SD controller. The FAT library is similar to the file handling functions in C with few syntax differences. Refer to the API for SD card in ECE423_API_REF.pdf document on learn.

Chapter 6

Performance Estimation

In order to optimize the system design, you need to collect essential performance metrics. In particular, how the various components of the system behave, and what resources they consume. Since our system is (relatively) simple, in terms of behavior we will only be interested in execution time and main memory (SDRAM) bandwidth, and in terms of resources, we will only consider FPGA utilization (but notice that power consumption is also paramount in most SoC).

6.1 Application Profiling

Application profiling is the dynamic collection of information on program execution, typically performed by instrumenting the code of the program - adding specific profiling instructions / system calls used to collect such information. While general purpose systems typically include a variety of advanced profiling tools (for example, perf in Linux), embedded systems often lack such functionality. Hence, in Lab1 you will write profiling code to measure the time required to execute certain functions (the functional blocks of the MJPEG423 decoder).

To measure time, we use the timestamp counter (a timer) added to the base system described in Chapter 5. Once you have a timestamp counter, you can use the timestamp API to measure time at the precision of a single clock cycle. The list of timestamp functions is provided in the timestampAPI.pdf file available on learn. If you want to measure the time required to execute a portion of code, simply place a call to alt_timestamp_start before before that portion of code, and a call to alt_timestamp after it. The alt_timestamp function will report the amount of clock cycles elapsed since the last call to alt_timestamp_start. You can use alt_timestamp_freq (or simply your knowledge that the timer is running at

125Mhz) to convert clock cycles in ns. Note that the default timer is configured with a 32bits counter. Therefore, beware that the counter will overflow after 2^{32} clock cycles.

6.1.1 Performing Meaningful Measurements

If you took ECE455, you learned that performing meaningful measurements (from a statistical perspective) is indeed very hard. You are free to incorporate the knowledge of ECE455 to design your measurements, but for those of you who did not take the course, here is a basic set of requirements.

The time required to execute a given functional block of the decoder might vary between frames. Hence, you should measure the time for each frame and compute some statistical metrics - at least average, minimum and maximum time.

The time required to execute the alt_timestamp_start and alt_timestamp functions is not zero. Hence, every time you are measuring the time to execute a portion of code, you are really also adding the execution time of the two timestamp function. To obtain better measures, first compute the "null execution" time, i.e., the time for an empty portion of code (alt_timestamp_start immediately followed by alt_timestamp). Then, derive the real execution time for a portion of code by subtracting the null execution time from the timestamp result for that code portion.

6.2 Memory Utilization

As it will be discussed at length in class, main memory bandwidth is often a very scarce resource, especially in multicore systems. Therefore, it is important to determine the memory utilization (in terms of amount of data read/written from/to main memory) of each functional block of the MJPEG423 decoder.

In general, since the Nios II core utilizes both an instruction and data cache, the overall memory utilization is a function of cache behavior, which can be difficult to evaluate (read also the explanation of data cache bypass in Section 7.4; to ensure that there is enough cache space to allocate frequently used data, it is beneficial to avoid loading in cache any data word that is only read or written once). However, in a (relatively) simple system like our decoder, most of the memory accesses are due to processing inputs and outputs. You can thus obtain an approximated bound to the amount of transferred data by summing the size in bytes of the inputs and outputs processed during each execution of a functional block. For most functions, such sizes can be statically determined. For example, the idct function always processes one 8x8 block of dequantized coefficients as input, and produces one 8x8 pixel color block as output. The ycbcr_to_rgb function takes three 8x8 pixel

color blocks as input, and (assuming you write the result directly to the pixel buffer in SRAM) does not need to write any output back to main memory. The lossless_decode is more complex since it depends on the size of the bitstreams, which is variable based on the frame; as explained in Section 6.1.1, you thus need to determine at least average, minimum and maximum size. Note that bitstream size information is provided for each frame in a MJPEG423 file.

6.3 FPGA Resource Utilization

In Lab2, you are asked to report the FPGA Resource Utilization of various hw blocks (Nios II, hw coprocessors). Note that there are three main resources on the Cyclone V FPGA:

- Logic Elements: there are 77k LE in the FPGA.
- Memory bits: there are 4,567,040 bits of internal RAM, organized as 4460 M10K RAM blocks of 10Kbits each.
- Embedded Multipliers: there are 300 18-bits embedded multipliers.

Unfortunately, the only way you can determine the usage of FPGA resource is after you compile the entire design in Quartus II. You can do so by selecting View Report under Fitter (Place and Route) (this is indeed because the exact resource usage depends on the place and route phase). You can determine the resource usage of a single hw component from the compilation report under Fitter, Resource Section, then Resource Utilization by Entity.

6.4 Memory Size

In Lab 3, you will further need to obtain a rough estimate of the memory footprint (space required in main memory) of each application in order to properly configure the multicore memory map. Luckily, this information can be easily acquired from the build summary produced every time an application is compiled and linked. Figure 6.1 shows an example build summary, with the derived program size highlighted in the red box. Note that this method only provides the sum of the required size of code and global data; in practice, extra space is consumed by the stack and heap of the application (see Section 8.2.1 for further details).

```
C-Build [slave_core]
--std=gnu99 -ffunction-sections -fdata-sections -Wl,--gc-sections  -EL -mno-hw-div -mhw-mul -mno-hw-mulx  -o obj/common/util.o common/util.c
nios2-elf-gcc: --gc-sections: linker input file unused because linking not done
Info: Compiling main.c to obj/main.o
nios2-elf-gcc -xc -MP -MMD -c -I../slave_core_bsp//HAL/inc -I../slave_core_bsp/ -I../slave_core_bsp//drivers/inc  -DSYSTEM_BUS_WIDTH=32 -pipe -D__hal__
-DALT_NO_C_PLUS_PLUS -DALT_NO_CLEAN_EXIT -DALT_NO_EXIT -DALT_NO_INSTRUCTION_EMULATION -DALT_USE_SMALL_DRIVERS -DSMALL_C_LIB -DALT_SINGLE_THREADED    -O3 -g -Wall
--std=gnu99 -ffunction-sections -fdata-sections -Wl,--gc-sections  -EL -mno-hw-div -mhw-mul -mno-hw-mulx  -o obj/main.o main.c
main.c: In function 'main':
main.c:38: warning: unused variable 'frame_counter'
main.c:41: warning: 'w_size' may be used uninitialized in this function
main.c:41: warning: 'hCb_size' may be used uninitialized in this function
main.c:69: warning: 'rgbblock' may be used uninitialized in this function
main.c:41: warning: 'wCb_size' may be used uninitialized in this function
nios2-elf-gcc: --gc-sections: linker input file unused because linking not done
Info: Linking slave_core.elf
nios2-elf-g++  -T'../slave_core_bsp//linker.x' -msys-crt0='../slave_core_bsp//obj/HAL/src/crt0.o' -msys-lib=hal_bsp -L../slave_core_bsp/  -Wl,--defsym,exit=_exit -msmallc
-Wl,-Map=slave_core.map   -O3 -g -Wall --std=gnu99 -ffunction-sections -fdata-sections -Wl,--gc-sections  -EL -mno-hw-div -mhw-mul -mno-hw-mulx  -o slave_core.elf
obj/common/tables.o obj/common/util.o obj/main.o -lm
nios2-elf-insert slave_core.elf --thread_model hal --cpu_name cpu_1 --simulation_enabled false --id 42 --sidp 0x1000120 --timestamp 1396310996 --stderr_dev jtag_uart_1
--stdin_dev jtag_uart_1 --stdout_dev jtag_uart_1 --sopc_system_name ECE423
Info: (slave_core.elf)  7760 Bytes program size (code + initialized data).
Info:                   2652 Bytes free for stack + heap.
Info: Creating slave_core.objdump
nios2-elf-objdump --disassemble --syms --all-header slave_core.elf >slave_core.objdump
[slave_core build complete]
```

Figure 6.1: Extracting Program Size from Build Summary

Chapter 7

HW Accelerators, Streaming Interface and DMA

As part of lab 2, you will need to design a new hardware accelerator to speedup the computation of the 2-dimension IDCT function. A simplified block diagram of the proposed architecture is presented in Figure 7.1. A DMA Core (DMA_Read) is used to read 8x8 blocks of dequantized coefficients (YDCAC, CBDCAC or CRDCAC matrix) from main memory, and pass it to the IDCT Accelerator through a *streaming interface*. The streaming interface is a simple synchronous interface designed for point-to-point connection of hardware IP blocks. The IDCT Accelerator computes the 2D IDCT on each incoming 8x8 block of dequantized coefficients, and output a corresponding 8x8 pixel color block (either for Y', CB or CR color component) to the second DMA core (DMA_Write) through another streaming interface. Finally, DMA_Write writes the 8x8 color blocks back to main memory in the corresponding Yblock, CBblock or CRblock matrixes.

Altera already provides the required IP block for the DMA Core (Modular Scatter-Gather DMA Core); your task is to design the IDCT Accelerator, connect all components and program the DMA to perform the required memory transfers. Note that to simplify your job, on learn we have provided a file idct_1D.vhd which implements a 1-dimensional IDCT in hardware. This is a pipelined implementation (4 stages) which is guaranteed to run at 125 Mhz on the target FPGA. Refer to Section 4.3 and to the code of the idct function in the reference implementation for how to implement the 2D IDCT based on the 1D version; since all mathematical operations are implemented in idct_1D.vhd, you will essentially need to code control logic to drive the 1D IDCT block, store intermediate results in hardware registers, and pass input/output blocks through streaming interfaces.

In the rest of the chapter, Section 7.1 provides an overview of the key details of the streaming interface. Section 7.2 discusses how to configure and program the DMA cores. Section 7.3 explains how to add a new IP component under Qsys with the required stream-

Figure 7.1: Basic Accelerator Design Block Diagram

ing interfaces. Finally, Section 7.4 discussed coherency between the Nios II cache and the DMA.

7.1 Avalon Streaming Interface

The Avalon-ST interface defines a simple, high-performance interface for unidirectional data communication between two hardware components. Before you start coding your hardware, you should make sure to read the full specification, which is available on learn. In this section, we summarize the key aspects of the protocol to get you started.

In Avalon-ST, communication is carried out between a data source device, implementing an Avalon-ST source interface, and a data sink device, implementing an Avalon-ST sink interface. In its simplest form, the interfaces include only 5 signals:

- Clock and reset; the interface is synchronous, hence the same clock must be used for source and sink. For the scope of the project, you should be able to employ the 125 Mhz clock which is also used for the Nios II processors. Signals are sampled on positive edges of the clock.
- Data. The data signal carries the information being transferred between source and sink. Every clock cycle, the signal transfers one or multiple *symbols*; each symbol is 8 bits. Hence, to transfer a word (32 bits) each cycle, the data signal should be configured with 4 symbols/clock cycle.
- Valid. This signal is produced by the source and qualifies data being transferred in the same cycle as valid.

Figure 7.2: Interface Transfer Diagram, READY_LATENCY = 0

- Ready. This signal is produced by the sink to mark the sink as ready to accept new valid data. The interface includes a configurable READY_LATENCY parameter, which determines how many clock cyles in advance the ready signal must be asserted before valid data can be produced by the source.

An example timing diagram is shown in Figure 7.2 for the simplest case of READY_LATENCY = 0 (refer to the document on Learn if you wish to implement a higher latency design, albeit this is not required). In this case, the source simply asserts valid and provides data every time there is available information to be transferred. If the sink asserts ready during the same clock cycle, then in the next clock cycle the source is free to send the next data item (if available). Otherwise, the source keeps transmitting the same data item. In other words, a transfer is considered successful only if both valid and ready are asserted during a clock cycle. As an example, in Figure 7, transfers are completed on the positive edges of clock cycles 3, 4, 7 and 8. Since on clock cycle 2 ready is not asserted, the source sends the same data again on cycle 3 (the same happens for cycles 6 and 7).

7.2 Modular Scatter-Gather DMA

The Modular Scatter-Gather DMA IP core can be used to implement the DMA functionalities required by the project. The core internally uses a FIFO queue of *DMA descriptors*; each descriptor contains all information required to perform one DMA transfer. To program the DMA core, a Nios II processor first prepares a descriptor data structure in memory, and then writes the descriptor information to the DMA core. Note that multiple descriptors can be sent to the DMA core while the core performs a DMA transfer; in this case, the core simply goes through the queue of descriptors, and completes each transfer one a time before moving to the next descriptor/transfer (if any remains in the queue). Additionally, the DMA core can send an interrupt to the the Nios processor to notify it that a transfer has been completed.

Multiple Port Front End

Export bonding interface

Expand Avalon-MM data for ECC

Number of ports: 3

Port	Type	Width	Priority	Weight	Allocated Write-Data FIFO	Allocated Read-Data FIFO
Port 0	**Bidirectional**	**32**	**3**	**0**	F0	F0
Port 1	**Read-only**	**32**	**2**	**0**	None	F1
Port 2	**Write-only**	**32**	**1**	**0**	F1	None

Figure 7.3: Memory Controller Port Configuration

To add a new Modular Scatter-Gather DMA core to the design, we can follow a similar sequence of steps as for the video_dma added in lab 1. The DMA core must connect directly to the DRAM memory controller (LPDDR2 RAM) to read or write to main memory. In lab 1, you already modified the controller configuration to add a read-only port (avl_1). Repeat the same steps to add a new write-only port to the memory controller (avl_2); Figure 7.3 shows the resulting configuration. As you add more components to the system, ensure you use the following configuration to minimize slowdowns due to memory interference:

- Nios II processors should be connected to the Bidirection Port 0 (avl_0). This ensures that the cores have the highest priority for access to main memory (as shown in the port configuration in Figure 7.3).
- Memory-read DMA cores should be connected to Read-Only Port 1 (avl_1), which has middle priority.
- Memory-write DMA cores should be connected to Write-Only Port 2 (avl_2), which has lowest priority.

Once you have added the write-only port, you can start adding the actual Modular Scatter-Gather DMA component. As discussed at the beginning of the chapter, you will need two of them for the basic accelerator design; more complex designs might require additional DMA cores. Figure 7.4 shows the suggested configuration for a write-DMA. Note that the DMA Mode is specified as Streaming to Memory-Mapped, that is, this core moves data from a streaming interface to main memory. For a read-DMA, the mode should be Memory-Mapped to Streaming. A in-depth description of the remaining parameters is available in the document *DMA Core* available on learn. To optimize memory performance, we recommend to perform transfers in multiple of 32-bits word size; hence, the Data Width of the core is set to 32 bits, and we specify Full Word transfer type. Ensure that the maximum transfer length is larger than or equal to the maximum size of any desired DMA data transfers.

To complete the Qsys configuration, connect all ports of the DMA core: the control ports (csr, descriptor_slave) are connected to the data_master port of the the cpu, the

Modular Scatter-Gather DMA
altera_msgdma
Details

DMA Settings

DMA Mode: Streaming to Memory-Mapped
Data Width: 32
☐ Use pre-determined master address width
Pre-determined master address width:: 32
Data Path FIFO Depth: 1024
Descriptor FIFO Depth: 16
Response Port: Disabled
Maximum Transfer Length: 512MB
Transfer Type:
◉ Full Word Accesses Only
○ Aligned Accesses
○ Unaligned Accesses
☑ Burst Enable
Maximum Burst Count: 128
☑ Force Burst Alignment Enable

Extended Feature Options

☐ Enable Extended Feature Support
☐ Stride Addressing Enable
Maximum Stride Words: 1
☐ Programmable Burst Enable

Streaming Options

☐ Packet Support Enable
☐ Error Enable
Error Width: 8
☐ Channel Enable
Channel Width: 8

Pre-Fetching Options

☐ Enables Pre-Fetching module
☐ Enable bursting on descriptor read master
Data Width of Descriptor read/write master data path.: 32
Maximum Burst Count on descriptor read master: 2

Figure 7.4: DMA Core Configuration, Avalon-ST Interface to Memory

interrupt port (csr_irq) is connected to the interrupt port (irq) of the cpu, and clock and reset are connected to the 125 Mhz source. The Memory Mapped Master port (either mm_read or mm_write) should be connected to the corresponding slave port of the LPDDR2 controller, as explained above. Finally, the Avalon-ST port (either st_source or st_sink) should be connected to the IDCT Accelerator.

The DMA Driver API is detailed in the referenced document (Page 33). For simplicity, this is the list of functions that you will need to use:

- alt_msgdma_open: this function opens the device and retrieves a device pointer.
- alt_msgdma_register_callback: this function registers a callback function (essentially, an interrupt handler) with the driver. The arguments to the function include the device pointer, a function pointer to the callback function, the context to be passed to the callback function (can be NULL), and a control field to specify additional callback control register information. A list of fields in the control register is reported in Table 25-8 in the referenced document, and corresponding defines for the flag masks are provided in the header file altera_msgdma_csr_regs.h in bsp_project/drivers/inc. Use value ALTERA_MSGDMA_CSR_GLOBAL_INTERRUPT_MASK to ensure that interrupts are enabled.
- alt_msgdma_construct_standard_mm_to_st_descriptor and alt_msgdma_construct_ standard_mm_to_st_descriptor: these functions are used to initialize the descriptor for a read/write transfer from/to main memory, respectively. The arguments to the function include the device pointer, a pointer to the descriptor to be initialized, the starting address in main memory for the transfer, the length of the transfer, and a control field to specify additional descriptor control register information. A list of fields in the control register is reported in Table 25-5, and corresponding defines for the flag masks are provided in the header file altera_msgdma_descriptor_regs.h in bsp_project/drivers/inc. In the context of the project, the only required mask is ALTERA_MSGDMA_DESCRIPTOR_CONTROL_ TRANSFER_COMPLETE_IRQ_MASK; passing this value in the control field informs the DMA core to invoke the callback function once the corresponding DMA transfer is complete.
- alt_msgdma_standard_descriptor_sync_transfer: this function writes a descriptor to the DMA core and starts a *synchronous* transfer, i.e., the processor that invokes the function is blocked until the transfer is complete.
- alt_msgdma_standard_descriptor_async_transfer: this function writes a descriptor to the DMA core and starts an *asynchronous* transfer, i.e., the processor that invokes the function is blocked until the transfer is complete.

- alt_msgdma_standard_descriptor_async_transfer: this function writes a descriptor to the DMA core and starts an *asynchronous* transfer, i.e., the processor that invokes the function is allowed to continue processing while the core performs the DMA transfer. If the processor needs to know when the transfer is completed, use the callback functionality.

7.3 Adding a New Component

To implement the required IDCT Accelerator, you will need to add a new hardware component to your design. Under Qsys, in the IP Catalog click New Component to start the Component Editor, the follow these steps:

- Under Component Type, specify a name for your component.
- Under Signals & Interfaces, you need to add all interfaces required for your component. As discussed, the proposed design requires a clock input, a reset input, an Avalon-ST source interface, and an Avalon-ST sink interface. Start by adding the clock, followed by the reset. Note that after adding each interface, you will need to add signals to it. The resulting signal list is shown in Figure 7.5.
- After adding the clock and reset inputs, add the required Avalon-ST interfaces. You can specify your preferred names for the interfaces; these will be used both in the Qsys connection diagram, and in your VHDL/Verilog design (the figure uses src for the sink port because it provides the input data for the implemented 2D IDCT hardware logic; similarly for dst being the output of the 2D IDCT). Ensure that the associated clock and reset are correct, as shown in the figure, and the interface uses 8 data bits per symbol. The "First Symbol in High-Order Bits" button allows to specify if you want data to be sourced/sinked in big-endian or little-endian format. We suggest to avoid using channels or error descriptors, and set the ready latency to 0. Each interface should have three signals, as shown in the figure: ready, valid and data. As discussed in Section 7.2, we suggest to use a Data Width of 32 bits, hence set the data signal to 32 bits. At this point you will notice that the block diagram reports a width of 8 bits for data; this is because we have not specified that we want 4 symbols per transfer. Unfortunately, there is no way to do this in the tool, hence an extra step is required.
- Click on Finish, generate the component, and go back to Qsys. At this point, you will notice that a file with the same name as your component and extension tcl has been created in your project directory. Edit the file and add the following line for each of the two Avalon-ST interfaces: "set_interface_property *name* symbolsPerBeat 4", where *name* is the name of the interface (dst and src for the example in the figure).

- After saving the tcl file, in Qsys click on File → Refresh. This will update the component description.
- Go back to the Component Editor, and from the menu open your component. Ensure that the data width is now set correctly.
- Finally, under Files, click on "create synthesis file from signal for synthesis file". This will automatically create a top-level file for your component with the correct input/output signals, based on the configured interfaces. If your implemented component requires multiple VHDL/Verilog files, add the additional files at this step.
- Click again on Finish to update all component information.
- Add the component to your design under Qsys and connect the Avalon-ST interfaces to the DMA cores.
- Re-generate HDL under Qsys and then synthesize your design under Quartus.

7.4 Cache Coherency

The Nios II/f employs a write-back data cache, but the Altera platform does not support cache coherency. If you took ECE429, you should realize why this is significant problem. Otherwise, consider the situation where a buffer is allocated in main memory to allow communication between a Nios II and a DMA. Since the data cache is write-back, whenever the Nios II writes to the shared buffer in main memory, the data is modified in cache but not in main memory, which instead is only updated when the corresponding cache block is evicted from the data cache. Therefore, if a DMA tries to read the buffer after the Nios II performed the write but before the cache block has been evicted, it will read old data - effectively, it does not see the write performed by the core. Note that in the reverse situation where the DMA writes to the buffer in main memory, the Nios II that reads from the buffer might fail to see the modified data simply because it loaded the cache block containing that portion of data in its local cache before data was modified.

To address this problem, most general purpose systems implement a *hardware cache coherency* mechanism, which ensures that the content of all local memories (caches) remain consistent. Since the Altera platform does not support such mechanism in hardware, you must address the issue in your software implementation. There are essentially two mechanisms to do so (additional details and function definitions are provided in the cache.pdf document on learn):

Figure 7.5: Component Editor for IDCT Accelerator

1. **Bypass the data cache:** the Nios II supports read and write instructions that bypass the data cache and access directly main memory. If you decide to go this way for a given data structure, then all accesses to that data structure must happen through bypass instructions. The Altera HAL provides suitable C macros that can be used to implement the bypass instructions: IORD_xDIRECT(BASE, OFFSET) and IOWR_xDIRECT(BASE, OFFSET, DATA), where x is either 32, 16 or 8, depending on the size of the data that you want to read or write. This functions work similarly to the IORD and IOWR macros that you should already be familiar with, except that the address used to read or write is BASE + OFFSET (with an IORD(BASE, REGNUM), registers are offset by 4bytes). This mechanism is suitable when: 1. each data word is only read or written once, hence there is no benefit in caching it; 2. or the data is allocated in a fast memory.

2. **Flush the data cache:** alternatively, you can use the data cache but selectively flush it (i.e., evict cache blocks containing shared data). If you decide to use this mechanism for a given data structure, a core must flush the data structure from its cache after it finishes modifying it. Similarly, a core must flush the data structure before it starts reading it. The Altera HAL provides two helper functions in sys/alt_cache.h: alt_dcache_flush_all () simply flush the entire data cache, while alt_dcache_flush (void* start, alt_u32 len) flushes the cache for a memory region of length len bytes, starting at address start. Note that executing flush_all is faster than the selective flush, but the downside is that you are evicting the entire content of the data cache, not just the shared data. You should use this mechanism when: 1. each data word is read or written multiple times by a single core before any other core needs to access that same data, hence caching it is beneficial; 2. and the data is allocated in a slow memory (such as SDRAM).

Chapter 8

Multicore Design with Nios II

This section explains how to create a multicore design using the Nios II and the Altera tools. Section 8.1 discuss how to synchronize cores, while Section 8.2 details how to create both the hardware and software design under Quartus II and the Nios II Software Development Tools.

8.1 Synchronization Primitives

Since multiple Nios II processors will share data, you need to properly handle concurrency in your code, in particular mutual exclusion and synchronization, as discussed at length in ECE254 (and equivalent courses).

Unfortunately, the Nios II ISA does not provide any atomic instruction, which is needed to implement concurrency mechanisms on multiprocessors. Therefore, Altera provides two IP components that can be used to implement concurrency mechanisms in hardware: the mutex component and the mailbox component. The mutex component implements the standard functionality of a mutex for mutual exclusion. The mailbox (simple) component again implements a standard mailbox functionality between two cores (sender and receiver).

To include a mutex or mailbox in your system, open Qsys and search for the corresponding component in the IP Catalog. A complete description of the mutex and mailbox API can be found in the mutex.pdf and mailbox.pdf files available on learn; most design can rely on the mailbox only. Note that the mailbox (simple) component implements a mailbox with a buffer of a single message; hence, a send operation fails if the buffer is full (the previously sent message has not been read yet). Synchronization on the receiver side can use either polling or interrupts. If you decide to use polling, you can disable the receiver interrupt line in the mailbox configuration window. We also suggest to disable

the sender interrupt line in the same screen. Make sure to connect the sender port of the mailbox to the sender core only, and the receiver port of the mailbox to the receiver core only; you can only have one sender core and one receiver core per mailbox, hence, you need a different mailbox for each sender-receiver pair.

Finally, keep in mind that as discussed in Section 7.4, the Nios II hardware does not support cache coherency. Hence, you must use the mechanisms described in Section 7.4 to ensure that data written by one Nios II core to main memory is correctly seen by another core.

8.2 Creating a Multicore Design in Quartus II

The facilities provided in the Altera Qsys make it reasonably easy to create a multicore design using the Nios II soft core. However, the Altera Software Tools do not allow you to create a natively multicore application using the Altera HAL API. Instead, you will need to create one application project + BSP for each core in your design. This means that the code for each Nios II must be compiled separately and downloaded to a separate region of memory.

There are three main steps needed to obtain a functional design: 1. determine the memory map for the multicore system (Section 8.2.1); 2. add and configure additional Nios II cores in Qsys (Section 8.2.2); 3. create the required projects in the Software Tools (Section 8.2.3).

8.2.1 Multicore Memory Map

The project employs a 512KBytes SRAM memory for code and a 512MBytes DDR2 RAM for data. When implementing a multicore design, both memories must be partitioned among the Nios II cores. In a typical system, the most complex step involves correct configuration of the linker on each core, but luckily, the Altera tools simplify this process.

The SRAM partitioning must be configured by changing the hardware design under Qsys. The second tab in the configuration window for each Nios II core shows the "reset vector" and "exception vector" addresses; these represent the address of the initialization code (code entry point) and of the interrupt vector table for that core. You should always set the least significant byte of their addresses to 0x00 and 0x20, respectively. After configuring the code entry point and exception address for all cores, the Altera tools are able to automatically infer the memory region allocated to each core. In particular, a core will use all memory starting from its code entry point and ending with either the code entry point of the next core, or the end of main memory itself. The memory region used

by each core must be sufficiently large to hold all code for the core itself. Note that failing to do so will cause an error during software build. The memory region size required to hold code can be inferred using the technique described in Section 6.4. Unless you use a large number of cores, dividing the SRAM in equal-size partitions should work for your project, but make sure to check.

DDR2 partitioning is configured trough the Software Development Tools. Open BSP editor for your BSP project and go to the Linker Script tab, shown in Figure 5.10. You can change the memory region associated with a core by modifying the size and offset of the lpddr2 linker region. Ensure that regions allocated to different cores do not overlap. There are three important considerations when configuring the region map. First, note that each core has a separate stack, heap and global data sections (bss, rodata and rwdata), which are placed in the memory region allocated to the core. Second, while each core is allocated a distinct region in main memory, the core is still able to read any address in main memory. In other words, a core can read data allocated in the memory region of another core, including dynamic data in the heap of the another core. The MJPEG423 sample decoder application allocates all required data buffers in main memory (Y/Cb/Crbitstream, Y/Cb/CrDCAC and Y/Cb/Crblock) during the initialization phase. Our suggested way to handle the buffers is to create all buffers using malloc on a single core. The mailbox functionality described in Section 8.1 can then be used to pass the addresses of relevant buffers to other cores.

Finally, the memory region used by each core must be sufficiently large to hold all data for the core itself; failing to do so might cause an error at run-time. The memory region size required to hold global data can again be inferred as discussed in Section 6.4. Unless you modified the program to include large data structures as local variables or recursive functions, the stack should not require more than a few kilobytes. The required heap size depends on the malloc calls performed by the code running on that core; as suggested above, if all mallocs are performed at the beginning of the program on one core, you can simply sum the size of all malloced data structures to determine the minimum required heap size. Please note that the video library also allocates data in the heap for each framebuffer, as discussed in Section 5.1. Unless you use a very large number of frame buffers, you should have plenty of space available in DDR2 for all cores, but make sure to check and to include some margins when configuring the system.

8.2.2 SOPC Configuration

Follow these steps to add and configure one or more Nios II cores to you project.

- Under Qsys, simply add a new Nios II peripheral. Make sure you use sequential numbers for all cores in the design, i.e. cpu_0, cpu_1, cpu_2, etc.

- If you want to print on your newly added core (highly suggested for debugging reasons), you must add an additional JTAG UART peripheral. Make sure the name you pick for the new peripheral is consistent with the name of the core, i.e., jtag_uart_0 for core_0, jtag_uart_1 for core_1, etc.

- Connect the ports for the newly added core so that they mirror the configuration for the existing core. In particular, note that you need to connect the data_master to lppdr2 and any peripheral used by the core, while instruction_master should be connected to SRAM. Memories and sysid peripheral should be connected to all cores that use them. Each mailbox / mutex should be connected to the cores that use it, based on their role for a mailbox (sender / receiver). No other peripheral should be shared among multiple cores. Each other peripheral should be connected to only one core. In particular: each button, timer, SD card PIO, DMA, and video peripheral must be connected to the data_master of one core (it does not have to be the same core for all peripherals though). Similarly, each interrupt line must be connected to a single core. Only the software running on the connected core should initialize the peripheral (if required) and access it at run-time.

- If you need additional peripherals (ex: an additional timestamp counter for the new core, since timers cannot be shared), simply add it to the project and connect it to the right core as explained.

- After completing all connections, configure each new core in the same way as the core already included in the project; in particular, use a Nios-II/f processor type and enable instruction and data caches. As discussed in Section 8.2.1, make sure to also properly configure the reset and exception vector for each core (you can only do so after connecting the core to memories).

An example of a resulting dual-core project showing the cores' connections is shown in Figure 8.1.

8.2.3 Multicore Software Configurations

As discussed, you need to create one application project + BSP for each core in the design. You can do so using the same procedure as for a single-core design. Note that on the first page of the wizard, the tools display the name of the core used for the project. In a single-core design you cannot modify this parameter, but now that you have multiple cores in your design, you can select the specific core that you want to use for this project. We suggest to name each application project based on the core it uses, i.e., decoder_0 for the application on core_0, decoder_1 for core_1, etc. Configure the BSP for each core similarly

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_125	Clock Source					
		clk_in	Clock Input	clk_125	exported			
		clk_in_reset	Reset Input	reset				
		clk	Clock Output	Double-click to export	clk_125			
		clk_reset	Reset Output	Double-click to export				
☑		clk_50	Clock Bridge					
		in_clk	Clock Input	clk_50	exported			
		out_clk	Clock Output	Double-click to export	clk_50_out_clk			
☑		cpu_0	Nios II Processor					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31	
		debug_reset_request	Reset Output	Double-click to export	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_0800	0x2010_0fff	
		custom_instruction_m...	Custom Instruction Master	Double-click to export				
☑		jtag_uart_0	JTAG UART					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_10e0	0x2010_10e7	
		irq	Interrupt Sender	Double-click to export	[clk]			1
☑		cpu_1	Nios II Processor					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31	
		debug_reset_request	Reset Output	Double-click to export	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2000_0800	0x2000_0fff	
		custom_instruction_m...	Custom Instruction Master	Double-click to export				
☑		jtag_uart_1	JTAG UART					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_10d0	0x2010_10d7	
		irq	Interrupt Sender	Double-click to export	[clk]			2
☑		sysid	System ID Peripheral					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_10d8	0x2010_10df	
☑		timer_1	Interval Timer					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_1000	0x2010_103f	
		irq	Interrupt Sender	Double-click to export	[clk]			4
☑		timer_0	Interval Timer					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_1080	0x2010_109f	
		irq	Interrupt Sender	Double-click to export	[clk]			0
☑		lpddr2	LPDDR2 SDRAM Controller with UniPHY					
		pll_ref_clk	Clock Input	lpddr2_pll_ref_clk	exported			
		global_reset	Reset Input	Double-click to export				
		soft_reset	Reset Input	Double-click to export				
		afi_clk	Clock Output	Double-click to export	lpddr2_afi_clk			
		afi_half_clk	Clock Output	Double-click to export	lpddr2_afi_h...			
		afi_reset	Reset Output	Double-click to export				
		afi_reset_export	Reset Output	Double-click to export				
		memory	Conduit	lpddr2				
		avl_0	Avalon Memory Mapped Slave	Double-click to export	[mp_cmd_cl...	0x0000_0000	0x1fff_ffff	
		mp_cmd_clk_0	Clock Input	Double-click to export	clk_125			
		mp_cmd_reset_n_0	Reset Input	Double-click to export				
		mp_rfifo_clk_0	Clock Input	Double-click to export	clk_125			
		mp_rfifo_reset_n_0	Reset Input	Double-click to export				
		mp_wfifo_clk_0	Clock Input	Double-click to export	clk_125			
		mp_wfifo_reset_n_0	Reset Input	Double-click to export				
		status	Conduit	lpddr2_status				
		oct	Conduit	lpddr2_oct				
		pll_sharing	Conduit	lpddr2_pll_sharing				
☑		key	PIO (Parallel I/O)					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_10c0	0x2010_10cf	
		external_connection	Conduit	key				
		irq	Interrupt Sender	Double-click to export	[clk]			5
☑		sram	Generic Tri-State Controller					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		uas	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2008_0000	0x200f_ffff	
		tcm	Tristate Conduit Master	Double-click to export	[clk]			
☑		sram_sharer	Tri-State Conduit Pin Sharer					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		tcm	Tristate Conduit Master	Double-click to export	[clk]			
		tcs0	Tristate Conduit Slave	Double-click to export	[clk]			
☑		sram_bridge	Tri-State Conduit Bridge					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		tcs	Tristate Conduit Slave	Double-click to export	[clk]			
		out	Conduit	sram_bridge_out				
☑		ledr	PIO (Parallel I/O)					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_1040	0x2010_105f	
		external_connection	Conduit	ledr				
☑		ledg	PIO (Parallel I/O)					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_1060	0x2010_107f	
		external_connection	Conduit	ledg				
☑		i2c_scl	PIO (Parallel I/O)					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_10b0	0x2010_10bf	
		external_connection	Conduit	i2c_scl				
☑		i2c_sda	PIO (Parallel I/O)					
		clk	Clock Input	Double-click to export	clk_125			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2010_10a0	0x2010_10af	
		external_connection	Conduit	i2c_sda				

Figure 8.1: Qsys Connections, Two Cores

to the single-core case, except that you need to modify the linker region map as discussed in Section 8.2.1.

To correctly start your multicore application, it is important to understand the precise behaviour of the platform. Whenever you download your hardware design or reset the board, all cores are first put in a idle state and wait for software code to be downloaded through JTAG. When you build a software project and run it on the Nios II hardware, only the core used with that specific project is initialized and starts running; the state of all other cores is left unchanged. Hence, if you want to run a system with n cores, you must manually select each of the n application projects you created and run that project to start the corresponding cores. Note that the Software Tools will open a separate Nios II console for each application project you run. Therefore, if you added a JTAG UART peripheral for each core, you can actually look at the output (printf) of each core individually, which greatly simplifies debugging. You also do not need to worry about mutual exclusion among cores for printing to their consoles: following the procedure described in Section 8.2.2, each core has its own JTAG UART peripheral, hence they can printf at the same time without problems (mutual exclusion is automatically solved in hardware by the JTAG UART modules for you).

We suggest the following initialization procedure:

- Use one core as your "master" core. This core should start-up the system by writing to mailboxes. You also probably want to use the master core to perform required initialization actions, such as create heap data structures (as described in Section 8.2.1), and to receive timer interrupts.
- All other "slave" cores should immediately wait on a mailboxes after starting.
- When you want to boot the system, first start all slave cores in any sequence. This ensures that the slave cores wait on the corresponding mailboxes. Then, start the master core. The master core can perform required initialization actions and then write to mailboxes, waking up the slave cores.

One final note is relative to running each application on the correct Nios II core. The Software Tools are sometimes not able to correctly relate each application project with the right core and JTAG UART peripheral. If the application fails to run, you can correct the problem by right clicking on its project and selecting Run As $->$ Run Configurations.... Make sure the correct project is selected in the box on the left, then switch to the Target Connection Tab on the right. In the tab, after refreshing connection to the board, you should be able to select and save the correct Nios II core and JTAG UART that you need to use for the project (if you followed our naming conventions and added the cores in order, you simply need to select cpu_i and jtag_uart_i for project decoder_i). If you do not see the

correct names of the instantiated Nios II cores and JTAG UART, press “Resolve Names” in the same tab to resolve the names based on the Qsys configuration.

Part III

Laboratory Projects and Assignment

Chapter 9

Lab1

9.1 Activities

Lab1 consists of the following activities:

9.1.1 Complete the HW platform.

Following Chapter 5, add all required hw components to the provided Nios II system using Quartus II and Qsys.

9.1.2 Execute the sequential MJPEG423 decoder

Make sure you understand the top-level behavior of the provided MJPEG423 decoder application; in particular, read the mjpeg423_decoder.c file. An in-depth explanation of the standard is provided in Chapter 4. Your goal is to execute the MJPEG423 decoder on a **single Nios II** (i.e., sequentially).

Note 1: the provided MJPEG423 sample application contains both the encoder and decoder. **You should not try to run the encoder on the Nios II** - instead, make sure you only execute the code of the decoder.

Note 2: the sample decoder reads a video file and outputs a sequence of bmp files, one for each frame, using the libbmp library. Since you want to output to the monitor rather than saving bmp files, you can strip this part away (including the entire library).

Note 3: since the Nios II is not fast enough to decode the video at a reasonable frame rate, the video will be shown in slow-motion; this is ok for Lab1 and Lab2.

Note 4: as discussed in Chapter 5, use double buffering to avoid screen tearing.

Note 5: you are only required to decode videos of size 640x480 pixels, since this is the size of the frame buffer used by the video controller.

Note 6: video files are in the root directory of the SD card. See Chapter 5 for a discussion of how to read from the SD card. The SD card contains video files of progressively larger duration and hence file size. You can begin testing the application using the first (smallest) file, but by the end of Lab 1 you should be able to decode all provided files.

Note 7: the sample application is coded in the C99 standard. By default, the Altera compiler uses the C89 standard. While you could alter the code to follow the C89 standard, it is way easier to change the compiler setting. In the Nios II Software Build Tool, select your software project, then go to Properties, Nios II Application Properties, and add *-std=c99* to the user flags. Furthermore, make sure that you turn on compiler optimization for your code, or it will run even slower! You can set the Optimization Level to Level 3 under Nios II Application Properties. You must do so separately for your application project and the bsp project, since they have a separate set of properties.

Note 8: you might need to perform byte-level operations and pointer casting. In case it is relevant for your code, note that the Nios II is a little endian architecture.

Note 9: as discussed in Chapter 5, the video controller uses a frame buffer in Y'CBCR color format; this is the same format as the output of the idct function in the decoder. On the other hand, the ycbcr_to_rgb function outputs a frame in RGB format. Therefore, modify the function to output the correct color format to the frame buffer (note that you still need a functional block, since you need to combine the output of the idct blocks for each of the three color components into a unique matrix of pixel colors, see Figures 4.4 and 4.5).

9.1.3 Implement Playback Control with Periodic Timer

Improve your application by implementing playback control and periodically displaying the frames. The system should accept the following inputs and behave as described below:

1. Pressing pushbutton 1 should cause the application to stop the current video (if any) and load the next valid video in the root directory of the SD card. A valid video has extension ".mpg". If the application has reached the last valid file in the root directory, pressing pushbutton 1 again should cause it to cycle back to the first valid file.

2. After a video has been loaded, pressing pushbutton 0 should cause the application to start playing the video. Pressing pushbutton 0 while the video is running should cause the application to pause/resume the video.

3. The time required to load a video must be short (below 1 second). It is not acceptable to read the entire video file into main memory when loading it.

4. Pressing pushbutton 2/3 should cause the video to skip behind/ahead by **roughly** 120 frames (5 seconds). If you skip ahead and the remaining number of frames in the video is less than 120, you should end the video. Similarly, if you skip behind when you are less than 120 frames into the video, you should restart decoding from the first frame. Finally, note that if the video has been paused (by pressing pushbutton 0), you still need to be able to skip behind/ahead while still pausing the video, and start playing from the new frame once pushbutton 0 is pressed again.

5. When the video ends, you should stop the video and wait for a pushbutton to be pressed.

6. While playing the video, you must display each successive frame at a **constant frame rate**; this must be achieved by switching the frame buffer (see Chapter 5) based on the periodic timer.

Note 1: to satisfy the requirement on load time, only read the header/trailer of the video file when loading it. Then, read the data of each frame as you are decoding the video. For Labs 1 and 2, a good solution is to read one frame at a time and then decode it before reading the next frame, just as implemented in the sample application.

Note 2: the sample MJPEG423 decoder reads the trailer of the file upon starting. As discussed in Chapter 4, the trailer contains the list of I-frames (frame number and position in the file) for the video file. You can implement the pushbutton 2/3 functionality by keeping track of the index of the current frame; when the user wants to skip behind/ahead, simply look for an I-frame that is roughly 240 frames ahead/behind the current frame and restart decoding from that frame.

Note 3: the base system described in Chapter 5 already includes a periodic timer (timer_1); ensure that you switch frame buffer at the beginning of each timer tick to satisfy the constant frame rate requirement. After switching frame buffer, you can then execute the decoder application to process the next frame; make sure you acquire a new frame buffer, as explained in Chapter 5, before writing to it at the end of the frame processing. The simplest way to synchronize the application with the periodic timer is to set a flag within the timer interrupt handler and have the application poll on the flag; make sure to check within the interrupt handler that the decoding of the previous frame has finished.

Note 4: since the application runs very slowly in Lab 1, you will have to set the timer period to a value much higher than 41ms (i.e., run the application much slower than 24 fps). This is fine for Lab1 and Lab2.

9.1.4 Profile the Application

Profile the decoder application based on the discussion in Chapter 6. In particular, you must profile the following functional blocks: 1. the code used to read the file from SD, one frame at a time; 2. lossless_decode; 3. idct; 4. ycbcr_to_rgb; 5. the code used to output the video, only if you did not implement it as part of ycbcr_to_rgb. For each block, report timing and memory utilization information.

For SD read and lossless_decode, you must distinguish between P and I-frames, since the two have quite different times. Similarly, compute the time and memory utilization for the Ybitstream first, and then for the Cbbistream and Crbitstream. You must report maximum, average and minimum times. For idct and ycbcr_to_rgb, there is no significant difference between frames and color components, so simply compute the time and memory utilization required to process all blocks in a frame.

Note 1: you will need to perform a similar profiling in Lab2 after changing parts of the application, so make sure your profiling code can be re-used later on. For example, you can enclose the profiling code within conditional compilation directives - note that the sample MJPEG423 application already uses conditional compilation to enable/disable debug information.

9.2 Deliverables

Demo: a demo will be held at the end of Lab1b session. You must first demonstrate the decoding application running on the Nios II without the periodic timer (i.e., decoding each frame one after the other), and then using the timer in conjunction with playback control. You must be able to decode all files provided on the SD card, but no profiling is required at this point.

Report: write a max 4 pages report detailing: 1. an overview of how you modified the sample application to execute on the Nios II, reading from the SD card, outputting to the video controller and handling playback control; 2. a table reporting all required profiling information; 3. a short discussion of the contribution of each lab member. A template describing the required format for the report will be provided on learn; you must follow the template when compiling the report. The mark breakdown percentage for the report is reported in Table 9.1.

Total Lab 1 Report	4%
Written Mechanics, Organization, Clarity	20%
Software Design	35%
Profiling Information	35%
Group Member Contribution	10%

Table 9.1: Mark Breakdown For Lab 1 Report

Chapter 10

Lab2

10.1 Activities

Lab2 consists of the following activities:

10.1.1 Create an Hardware Accelerator for IDCT

Following the discussion in Chapter 7, create a custom hardware accelerator to speed up the execution of the idct function. To help you in this activity, we provide on learn a VHDL module (idct_1D.vhd) which performs the 1D IDCT (see Chapter 4 for details on how the 2D IDCT is computed based on the 1D IDCT). Connect the accelerator with required DMA cores and execute the sequential decoder application using the accelerator to compute the 2D IDCT transform for each coefficient block.

Note 1: Read the comments provided with the idct_1D.vhd file to understand the interface to the module. It takes as input 8 coefficients and returns 8 coefficients, each of which is 16 bits. The module can be configured to run either Pass 1 or Pass 2 (it simply changes the descaling and normalization at the end). The module is implemented as a 4-stages pipelined designed, guaranteed to run at 125 Mhz on the target board.

Note 2: The provided idct_1D is written in VHDL. You are free to use either VHDL or Verilog to implement the 2D design, but make sure to use synthesizable register-transfer level syntax.

Note 3: Before adding the accelerator to your design, you might want to test the functionality of the DMA cores. For example, you could directly connect the source streaming interface of the read-DMA to the sink interface of the write-DMA and perform a main memory to main memory transfer.

Note 4: You might also want to simulate your VHDL/Verilog code before synthesizing the design. A set of suitable test vectors are provided on Learn. A simple-to-use simulator is available online at: `http://www.edaplayground.com/home`, but you can use industry-level simulators (ex: ModelSim) if you are familiar with them.

Note 5: In lab 2 you are only required to run the application sequentially; hence, the CPU does not need to perform any additional frame processing while the accelerator transforms pixel blocks. However, you must still allow the two DMA cores to transfer data simultaneously. A simple strategy to achieve this is to program each DMA core to perform three transfers, one for each color component. You can then stall the processor waiting for the last write transfer to finish by either waiting on an interrupt, or using a synchronous transfer.

Note 6: There is a non-negligible overhead involved in initializing DMA descriptors and passing them to the DMA core. Hence, try to perform large-size DMA transfers as much as possible.

10.1.2 Optional: Create an Hardware Accelerator for Color Combiner

Create a second accelerator to implement the color combiner functionality (i.e., the functionality of the previous ycbcr_to_rgb function, modified in lab 1 to combine blocks of individual color components into pixels in Y'CBCR format), then execute the sequential decoder application using the accelerator in place of the software implementation. **This activity is optional, and will not be graded in lab 2**. However, it can significantly speed up the application and simplify the multicore schedule in lab 3.

Note 1: Think carefully of how to implement this functionality. Since you want to be able to write the result to the frame buffer in main memory with the minimal number of DMA transfers, you will likely need to buffer into the hardware component at least 8 rows of pixels. The targeted FPGA has enough on-chip memory to support this.

10.1.3 Profile the Application and the Platform

Profile again the application in the same way as in Lab1, for the same Nios II configuration, but this time after adding the hardware accelerator. You must report the new time for the idct functionality.

Then, using the technique described in Section 6.3, derive the FPGA resource utilization for the following components: 1) Nios II processor(s); 2) DMA core(s); 3) hardware accelerator(s).

10.1.4 Optimize Application Period Based on I-P frame Sequencing

Based on the obtained profiling information, determine the minimum period at which the sequential application can be run, and modify your periodic timer correspondingly. Since I-frames take longer than P-frames, you will need to change the video driver configuration to increase the number of frame buffers.

10.2 Deliverables

Demo: a demo will be held at the end of Lab2b session. For the MJPEG423 application, you must demonstrate the use of the hardware accelerator while still supporting playback control and being able to decode all video files at a constant frame rate using the timer. You must also show that you are able to reduce the period by increasing the number of frame buffers.

Report: write a max 4 pages report detailing: 1. the design of the hardware accelerator; 2. a table reporting all required profiling information and FPGA utilization; 3. a discussion of how you determined the minimum period and number of frame buffers; 4. a short discussion of the contribution of each lab member. A template describing the required format for the report will be provided on learn; you must follow the template when compiling the report. Submit all your code (including the profiling code) together with your report. The mark breakdown percentage for the report is reported in Table 10.1.

Total Lab 2 Report	4%
Written Mechanics, Organization, Clarity	20%
Accelerator design	40%
Period computation	15%
Profiling Information	15%
Group Member Contribution	10%

Table 10.1: Mark Breakdown For Lab 2 Report

Chapter 11

Design Assignment

Before your group starts implementing the multicore design required in Lab 3, it is important that you determine how to configure the hardware platform and map the functional blocks of the MJPEG423 decoder on the platform. In particular, you should perform the following activities.

- Platform configuration: determine the number of Nios II cores you want to use and their configuration, as well as the number and configuration of DMA modules / HW accelerators for IDCT.
- Mapping and scheduling: determine which functional blocks are executed on which Nios II core or HW accelerator, as well as the order (schedule) in which the blocks are executed. This also involves selecting the frame rate at which you want to run the decoder (ideally, 24 fps).
- Feasibility analysis: determine whether you can successfully run the designed hardware/software system while meeting all constraints. In particular, you should determine whether you have enough FPGA resources (logic elements, DPS units, internal RAM blocks, as well as space in the SRAM and DDR2 memories), and whether you can feasibly execute the application based on the chosen schedule (i.e., can you finish decoding frames in the allotted time?).

In general, we do not dictate a strict methodology for designing the platform. While during the course we cover design exploration algorithms, the effort required to learn or implement an automated selection algorithm is higher that designing the platform "by hand" for a project of this complexity. Instead, you should use sound engineering principles to pick the number of cores, configure the Nios II and assigning functional blocks to cores.

The remaining sections in this chapter provide additional information related to scheduling, synchronization and overheads.

11.1 Application Scheduling

A description of the dependencies among functional blocks of the application is provided in Section 4.8. Based on the discussion and application of Task Graph Scheduling seen in class (Part III of the course), and assuming you determined a platform configuration, your goal is to map functional blocks to cores and create a schedule for the application. You should base your schedule on the profiling information that you gathered in Lab 1 and 2, incorporating overheads if needed as described in Section 11.1.3.

If you plan to run the application at 24 frames per second, you effectively have 1000/24 = 41.67ms to process each frame on a given core; accounting for overheads, the functional blocks executed on the core should likely take less than 40ms. The only possible exceptions are I-frames; an I-frame can take longer than the allocated 41.67ms as long as the schedule can "catch up" by running shorter P-frames following the I-frame. Following the scheme employed in Lab 2, you should thus design your schedule to incorporate multiple frames, and increase the frame buffer size so that multiple P-frames can be processed in between displaying successive frames. Please note that the worst case pattern of I and P-frames encountered in any of the provided videos is the following: I, P, P, I, followed by 23 more P-frames (while one could compress a video to have more frequent I-frames, you can assume the provided worst case pattern while designing your schedule). Note that we do not suggest drawing a completely different schedule for 27 consecutive frames - instead, try to draw a schedule with a small number of frames (possibly fixing the execution order for every I-frame and every P-frame), and then simply assume it is periodically repeated. To determine the makespan, we suggest to use: 1. Worst-case profiling information for the first I-frame. This is because you could jump to any I-frame in the video, hence you could suffer the maximum processing time at the beginning of the sequence; 2. Average case for successive frames. If you use average values, ensure that you have sufficient margins (10-20%) to avoid missing frames while processing the video.

You are free to decide how to break down the functional blocks into tasks. In the example seen in class, we considered each functional block for an entire color component of a frame as a single task. However, as discussed in Section 4.8 you can further break down the functional blocks at the level of groups of 8x8 pixel blocks (for example, one row of blocks) if it helps scheduling the application.

11.1.1 Buffering

As discussed in class, you might need to duplicate certain buffers between functional blocks once the makespan of a frame becomes larger than the period. Let Δ be the makespan of a frame, computed from the time when the frame should start processing (according to the period) to when the frame is output to video; then the worst-case buffer depth is $\lceil \Delta / P \rceil$,

where P is the period. Note that for the frame buffer, you will need $\lceil \Delta/P \rceil + 1$ buffers in the worst case, since the currently displayed frame accounts for one extra buffer as it needs to remain in memory while displayed. You can use this strategy to determine the minimum number of frame buffers that you need for your schedule to work.

Finally, pay special attention to the YDCAC, CBCDAC and CRCDAC data structures. As detailed in Chapter 4, the lossless_decode function for a P frame takes each of the three data structures as input and modifies its values. The modified values are then used as input by both the idct function for the same frame, and by the lossless_decode function for the next frame. Should you need a buffer depth of two (or larger), you would have to either modify the lossless_decode function to read one buffered copy of YDCAC and write to the other copy of YDCAC (the same holds for CBCDAC and CRCDAC), or make a copy of the data structures after processing the lossless_decode. In either case, such operations involve a large overhead; since you are basically making an extra copy of the data structures that are input to the idct, the overhead is similar to the time to process the idct for the entire frame using DMA. Hence, try to avoid duplicating the three data structures. Since the HW idct accelerator should be faster than the lossless_decode on the Nios II, you can for example divide the frame in two sets of 8x8 blocks and pipeline the lossless_decode on one half of the image with the idct on the other half.

11.1.2 Synchronization

While executing the various functional blocks on different cores, you must ensure that the blocks are run in the correct order. While there are multiple possible ways to synchronize the cores based on the discussed model, we suggest to use the following scheme:

- Let the "master" core be the core that receives the periodic timer interrupt. Switch the frame buffer every timer interrupt to ensure that the frame rate is constant. We also suggest implementing the basic playback control logic in the master core, so that the execution on the other slave cores can be paused / resumed after synchronizing with the master.
- To synchronize Nios II cores and DMA, use asynchronous transfers - in this way the DMA can continue transferring data while the core executes a different functional block. Use interrupts / callback functions to check that DMA transfers have completed.
- Use a mailbox to activate a task that depends on another task running on different core. As an example, Figure 11.1 shows a basic scheduling scheme (not suggested) between two cores. Note that each idct (ID) task depends on the lossless_decode (LD) task for the corresponding color component, while the color combiner (CC) depends

Figure 11.1: Example Synchronization

on all idct. To enforce the precedence constraints, we can implement two mailboxes: one for synchronization from core 1 and core 2, and one for synchronization from core 2 to core 1. As an example, before executing the ID Y block, core 2 should wait on the first mailbox. Conversely, core 1 can write to the same mailbox after completing the execution of the LD Y block. The mailbox message can conveniently contain the address of the data structure (buffer) that ID Y needs to operate on. Finally, if you use a mailbox to synchronize among multiple task pairs, keep into account that the mailbox contains a single message. Hence in the figure above, core 1 cannot send a message to core 2 before core 2 reads the previous one; hence for example, LD Cr cannot start before ID Y starts and consumes the first message sent after the completion of LD Y.

11.1.3 Overheads

It is important to take into account that the execution of individual functional blocks on the multicore platform can be slower than on the single core platform due to additional sources of overhead. Therefore, you should make sure to leave enough margins in your computed schedule to finish running your frame sequence within the allocated time.

The first main source of overhead is due to synchronization operations, i.e., reading and writing to mailboxes and preparing and sending DMA descriptors. If you only access mailboxes and program the DMA units a small number of times per period, you can likely safely ignore this overhead. However, if you decide to synchronize the application at a smaller granularity (ex: 8x8 pixel blocks), the number of mailbox and DMA API calls might become much larger. In this case, you should: 1. profile the time taken to read to / write from a mailbox on your system and / or synchronize with the DMA; 2. add the time required for synchronization to the maximum running time of each functional block; 3. compute the application schedule based on such inflated time.

The second main source of overhead is memory contention; all cores share SRAM for code, and all cores and DMA share the DDR2 RAM for data. To minimize the effects of memory contention, in Labs 1 and 2 the DDR2 memory controller has been configured with multiple memory ports. As you connect additional Nios II cores, ensure that the

data_master of all cores is connected to Port 0, which has been configured with the highest priority. This ensures that cores have higher priority for access to main memory compared to DMA components. Under the suggested configuration, cores should suffer very limited slowdown compared to the single-core case. On the other hand, DMA components are highly affected by other DMA components; a pair of read and write DMAs are generally sufficient to saturate the available memory bandwidth, hence, increasing the number of DMAs will not improve performance.

Chapter 12

Lab3

12.1 Activities

Lab3 consists of the following activities:

12.1.1 Create the multicore HW platform

Following Chapter 8, create a multicore Nios II system using Quartus II and Qsys to execute the parallel MJPEG423 decoder. Configure the system based on the result of the design assignment in Chapter 11.

12.1.2 Execute the parallel MJPEG423 decoder

Parallelize the MJPEG423 application and run it on the multicore Nios II platform. Your application should be able to decode a MJPEG423 file at a **constant frame rate**.

Note 1: before starting to parallelize the MJPEG423 application, make sure to properly test the configured hardware system with a simple application to ensure that all connections are correct.

Note 2: as part of the Design Assignment, you must make certain assumptions on the execution time of the various functional blocks to guarantee that a sequence of frames can be completed within the allotted time. Since the multicore platform will introduce additional timing overhead as explained in Section 11.1.3 and your design might use average, rather than worst case timing, you must ensure that when you run the application you can indeed meet the timing constraints. To do so, you must check that whenever a timer interrupt arrives the next frame is available (i.e., you can successfully switch the frame buffer to the next frame).

12.2 Deliverables

Demo: a demo will be held during the Lab3b session. To meet functional requirements, your design must run on at least two cores at a constant frame rate. As in the previous labs, you need to support playback control and be able to decode all video files. The design should be reasonably optimized, i.e., the schedule should be reasonable based on your chosen number of cores.

Furthermore, Lab 3 includes an additional grading component based on the performance of your implemented system in terms of frame rate; to meet non-functional requirements, you should come as close as possible to (but no higher than) 24 fps.

Report: your final report should include a complete description of your design and implementation of the multicore Nios II platform and parallel application. In particular, the report should include the following sections: 1. a detailed description of the activities carried out during the Design Assignment, as discussed in Chapter 11. In particular, you should discuss how you determined: A. platform configuration and mapping; B. application scheduling and synchronization, including required buffer sizes; C. feasibility analysis (ensuring that you have enough resources to implement the hw/sw system). 2. A discussion of the implementation of the platform. In particular, you should discuss any difference between the analysis results and what you obtained from the implementation. 3. A short discussion of the contribution of each lab member. A template describing the required format for the report will be provided on learn; you must follow the template when compiling the report. The mark breakdown percentage for the report is reported in Table 12.1

Note 1: Meeting 24 fps during the demo guarantees the maximum performance mark. You can still receive a good performance mark without reaching 24 fps if you show that you can achieve a clear speedup over the sequential version of the application demonstrated in Lab 2.

Note 2: If you implemented the optional color combiner hw accelerator in Lab 2, you might be able to reach 24 fps with a very simple schedule on 2 cores. Otherwise, you might need a more complex schedule or a third core for the color combiner. In any case, ensure that the report contains proper analysis and configuration. For example, do not configure the system to use 24 frame buffers if your schedule still works with just 4 or 5.

Total Lab 3 Report	9%
Written Mechanics, Organization, Clarity	20%
Design: Configuration, Mapping, Scheduling and Synchronization	40%
Design: Feasibility Analysis	15%
Implementation and Results	20%
Group Member Contribution	5%

Table 12.1: Mark Breakdown For Lab 3 Report