ECE423 Embedded Computer Systems

Lab Manual

by

Rodolfo Pellizzoni

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, 2014

Contents

List of Tables

List of Figures

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 DE2 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 ECE224/324/325 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 coprocessors: 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 hw coprocessors 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.

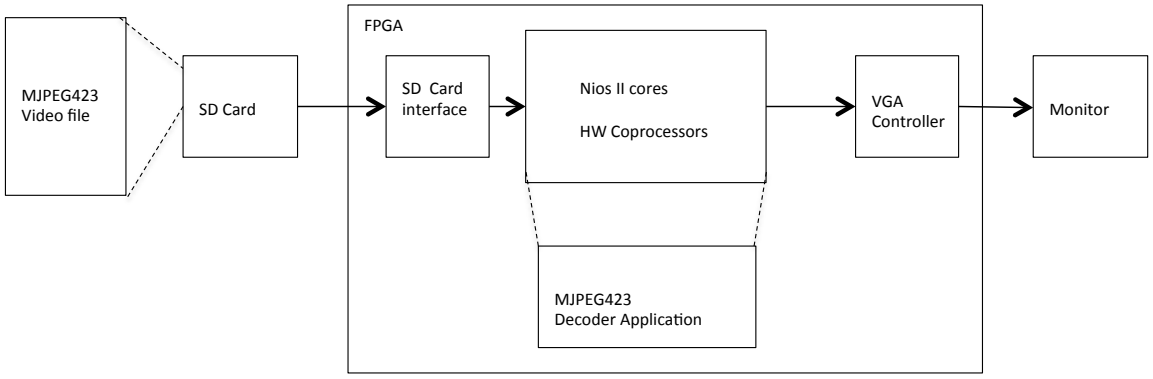

Figure 1.1: Decoder System Block Diagram

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 at the end of this activity 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 DE2 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 hw coprocessors to speed-up the execution of critical parts of the applications. 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 coprocessors); 3. integration (how to put hw and sw together, profile, run and synchronize the application).

Submit lab groups	Monday Jan 12 8:30AM
Lab1a session	M-T-W Jan 12-13-14
Lab1b session and demo	M-T-W Jan 26-27-28
Lab2a session	M-T-W Feb 9-10-11
Lab2b session and demo	M-T-W Mar 2-3-4
Lab3a session and discuss design	M-T-W Mar 16-17-18
Lab3b session and demo	M-T-W Mar 30-31 Apr 1
Lab 1 mini-report and code due	Monday Feb 2 8:30AM
Lab 2 mini-report and code due	Monday Mar 9 8:30AM
Final report and code due	Monday Apr 6 11:59PM

Table 1.1: Course Timeline

Total Lab	35%
Lab1 demo	4%
Lab1 report	4%
Lab2 demo	4%
Lab2 report	4%
Lab3 demo	4%
Lab3 performance	3%
Final Report	12%

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 each group must satisfy the following requirement: at least one student in the group must have attended and passed ECE254 / 354 / MTE 241 (or equivalent). Please note that the rule is in place to ensure that you do not run into issues in Lab 3. It is also advisable that at least one student in each group is knowledgeable about coding digital logic using VHDL; if no student in your group attended ECE 327, please make sure to revise the material in ECE 124.

A sufficient number of groups for all students in the course has been created in Learn. You can form your group by simply joining one of the groups on Learn. Note that groups are named and numbered based on the assigned day of the week for lab sessions. **Make sure you form your group by Monday Jan 12 at 8:30AM.** If you cannot find a group by the stated deadline, please send an email to the Lab 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 7:00PM - 10:00PM in E2 2364. The lab is 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/code submission. Lab demos will be conducted during the final hour of the scheduled lab session, unless a group wants to perform the demo earlier. The report and code must be submitted through

Learn; there are separate dropboxes for each. **Make sure you compress all your code into one archive to avoid submitting a large number of files.**

Grace Days There are four grace days that can be used for late submissions of lab reports and code. 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.

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 / 324 / 325 lab project. In particular, we assume that you should be able to create a custom microprocessor design on DE2 using Quartus consisting of a Nios II processor, SD interface, SDRAM memory controller, timer and required PLL logic. 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 a second memory controller, 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 interface HW coprocessors with the Nios II, 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

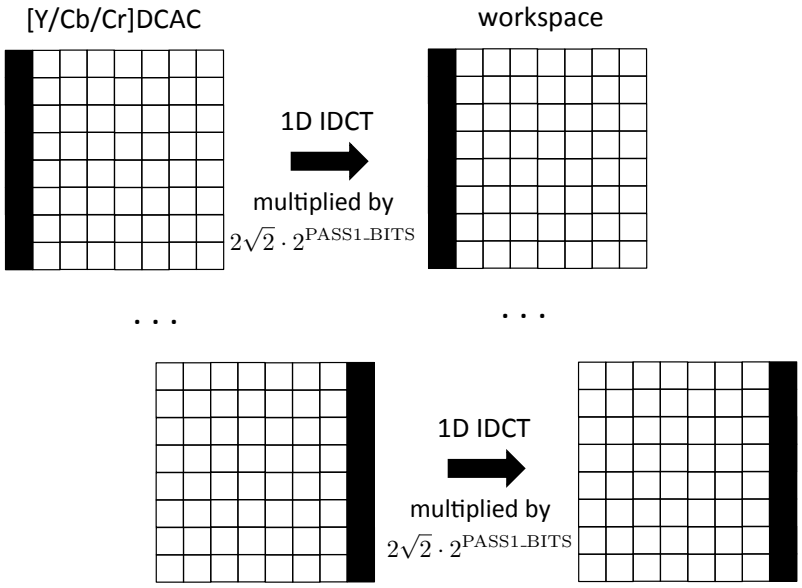

Figure 4.2: 2D IDCT Pass 1: Column Processing

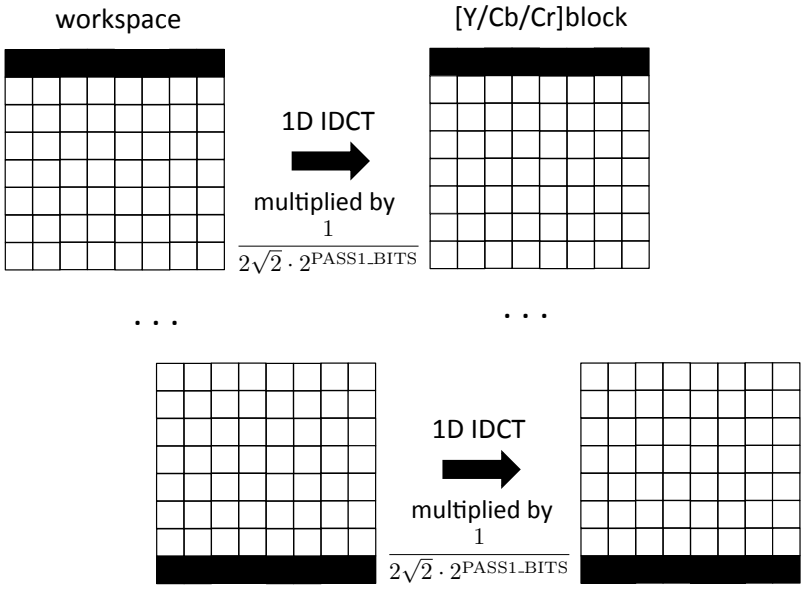

Figure 4.3: 2D IDCT Pass 2: Row Processing

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 within the payload 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 a factor of 7-8 times). 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 with/without hw coprocessors). 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 IDCT nodes on 2 Nios II cores). This

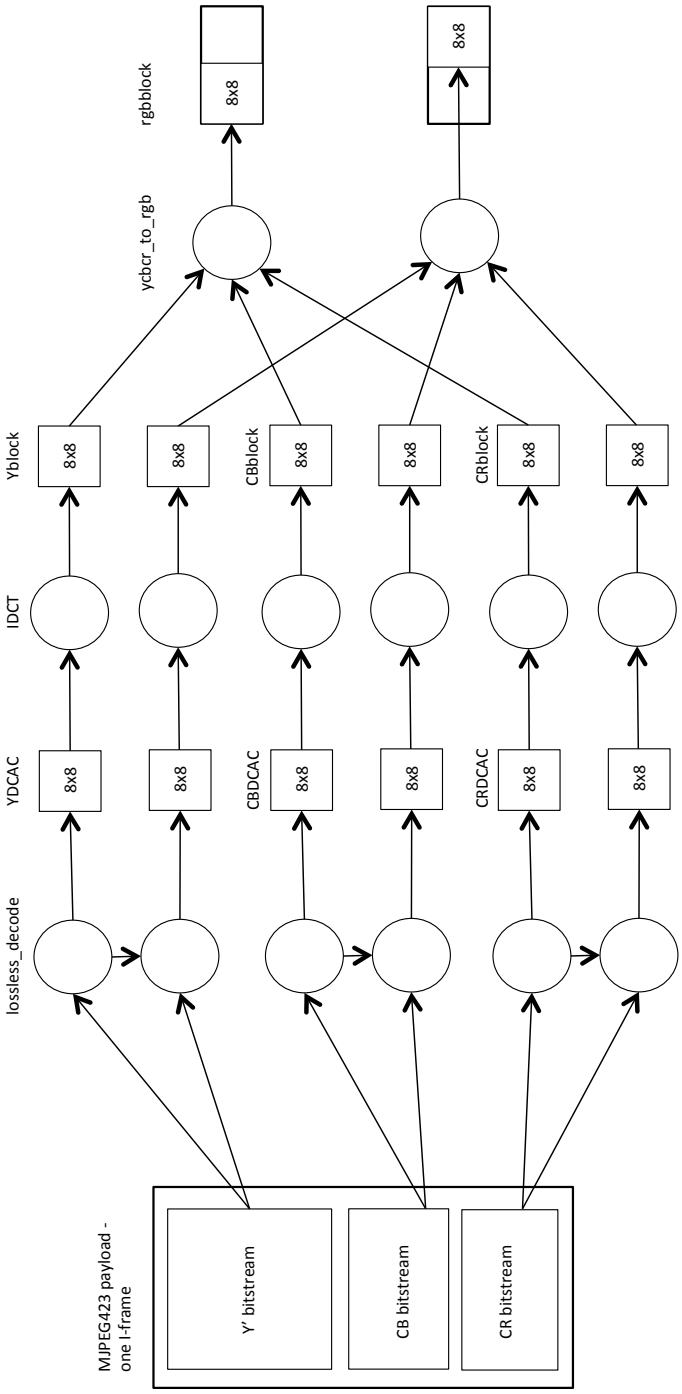

Figure 4.4: I-Frame Processing DAG

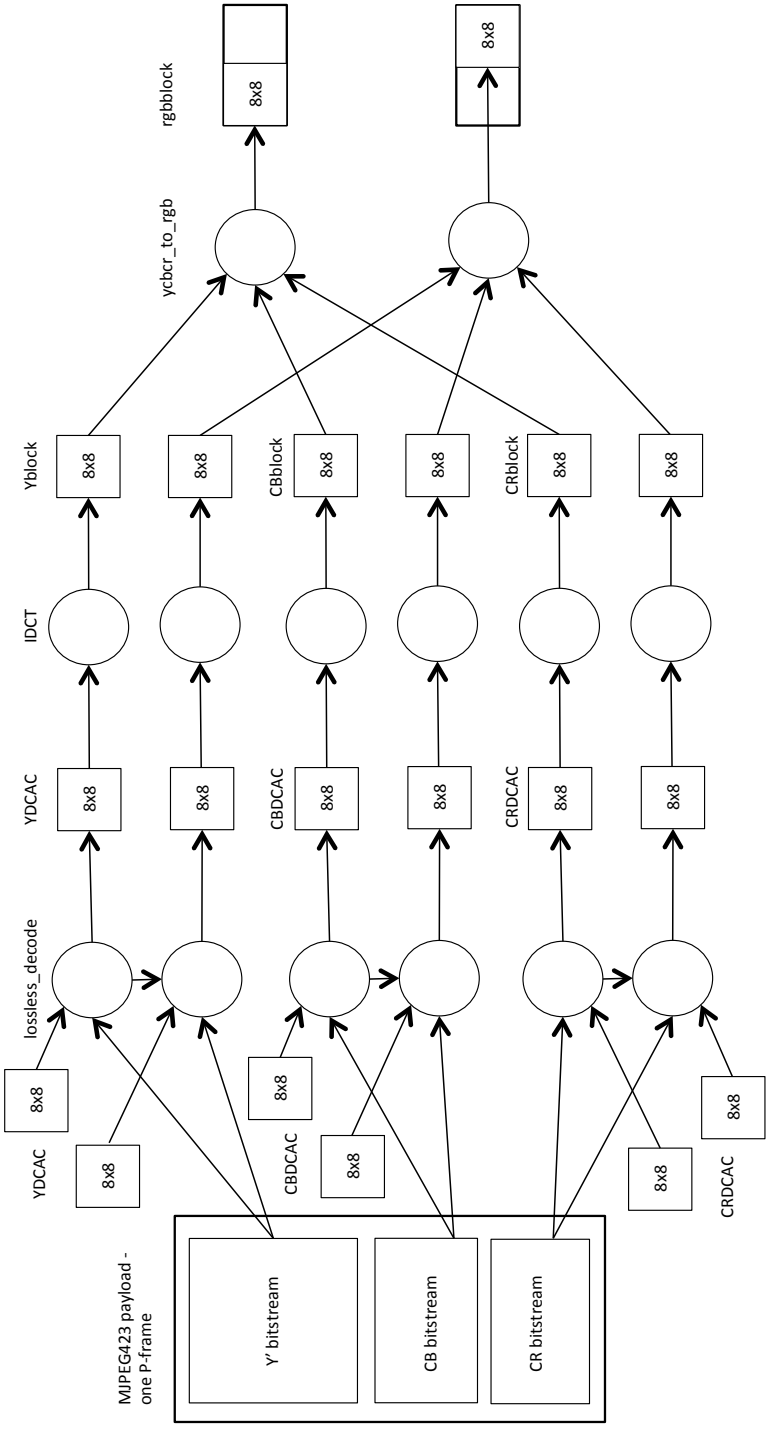

Figure 4.5: P-Frame Processing DAG

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 units and hence of FPGA area 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 SOPC builder 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.

Create a new Quartus II project. As a reminder, the DE2 board uses a EP2C35F672C6, speed grade 6 device (FBGA, 672 pins). Then create the top-level design (the ECE224 used a Block Design File, but you can use a VHDL/Verilog file to assign pins if you are more comfortable with it) and start SOPC builder. Make sure that Device Family is set to Cyclone II and the clock (clk_0) is 50Mhz. Also, **make sure to assign different names to the project/top-level entity and to the SOPC builder module.** For example, if you call the project mpeg423, you can use mpeg423_controller for the SOPC builder module.

In SOPC builder, add the following peripherals to the design, similarly to ECE224:

- Nios II processor (under Processors). In the wizard, select the NiosII/f version, and ensure that Hardware Multiply is set to Embedded Multipliers. This ensures that the MUL instruction is implemented in hardware. You can leave all other options unchanged, but you might want to consider increasing the debugging level on the 5th page to simplify debugging in the Nios II Software Build Tool (this will significantly increase the use of FPGA resources, which is not an issue in Lab1 and 2 but might constrain your design in Lab3). Also, remember to set the reset and exception vectors after you add main memory (SDRAM) to the system.

- System ID (under Peripherals, Debug and Performance). Remember that this peripheral must be renamed to sysid.
- JTAG UART (under Interface Protocols, Serial). You can use default settings.
- SDRAM Controller (under Memories and Memory Controllers, SDRAM, SDRAM Controller). Select Custom in the presents drop down box. Set the data width to 16, all other settings can be left as is.
- Interval Timer (under Peripherals, Microcontroller Peripherals). You can use default settings.
- PIO for the Push Buttons. As in ECE224, you should configure them as: 4 bits inputs, synchronous capture on any edge, and generate edge interrupt. They should be connected to the KEY3..KEY0 pins.

You are free to add other components (ex: LCD, LED, Seven Segment, switches) if they help you debugging, but they are not needed.

Next, you will add the components required to handle the VGA output. 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 VGA IP cores work is provided in Section 5.1.

- We need a new memory to hold the framebuffer. Add a SRAM/SSRAM Controller (under University Program, Memory). As with all other cores, ensure that the DE-series board selected is DE2. Select Use as a pixel buffer for video out.
- At this point, make sure to Auto-Assign Base Address so that all memories are assigned base addresses.
- Add a Pixel Buffer DMA Controller (this core and all following cores are under University Program, Audio and Video, Video). Set the addressing mode to consecutive. Set the default buffer start address to the base address of the SRAM memory. Set the default back buffer start address to the default buffer address + 0x40000 (or 256KBytes, which is half the size of the SRAM). Use a frame resolution of 280x200, and a color space of 24bit RGB. After you close the wizard, connect the avalon_pixel_dma_master connection of the Pixel Buffer DMA to the avalon_sram_slave connection of the SRAM.
- Add a Clipper. Set the incoming frame resolution to 280x200. Set all reduce frame size fields to 0. In enlarge frame size, select 20 (rows or columns) to the left, right, top and bottom. Finally, set the pixel format to 8 color bits and 3 planes. You should see an info tab telling you that the resolution is changed from 280x200 to 320x240. After

closing the wizard, connect the avalon sink of the clipper to the avalon_pixel_source of the Pixel Buffer DMA. This ensures that the Pixel Buffer DMA sends data to the Clipper (and should remove some error messages from the info tab at the bottom of SOPC).

- Next add a RGB Resampler. Use 24-bit RGB as the input format and 30-bit RGB as the output format. Connect the avalon sink of the RGB Resampler to the avalon source of the Clipper.

- Next add a Scaler. Use a scaling factor of 2 (both width and height), incoming resolution of 320x240, and pixel format of 10 color bits and 3 planes. The info at the bottom should tell you that the resolution is now changed from 320x240 to 640x480. As you likely figured out by now, connect the avalon sink of the Scaler to the source of the RGB Resampler so you continue the chain.

- The VGA Controller needs a 25Mhz clock. Under Clock Settings in SOPC, click on Add and configure the new clock (clk_1) to be an external clock with 25.0Mhz frequency.

- Since the Scaler uses a 50Mhz clock, we need a dual clock buffer to communicate with the VGA. Add a Dual-Clock FIFO to the project. As in the Scaler, use a pixel format of 10 color bits and 3 planes. Connect the avalon sink of the Dual-Clock FIFO to the source of the Scaler. Make sure that under clock, the avalon sink uses clk_0 (50Mhz). Change the avalon source of the Dual-Clock FIFO to use clk_1 (25Mhz).

- Finally, add the VGA Controller. No options needed (the VGA Controllers requires a 640x480 RGB image with 10 color bits and 3 planes). Connect the sink of the VGA Controller to the source of the Dual-Clock FIFO, and change it to clk_1.

Finally, you need to add an SD Card Controller. Contrary to ECE224, we use a dedicated hardware peripheral to speed up read operations from the SD Card. The controller is located under University Program, Memory. There are no relevant settings.

After you finish adding all the components, remember to and assign base addresses and IRQ. At this point, you should have no remaining error messages in the information tab at the bottom of SOPC (info messages are fine - they tell you the configuration of the VGA chain). Figure 5.1 shows the resulting configuration (yours might have a different order or different addresses, which is fine). You can now generate the SOPC block.

Once you return to Quartus II, add the SOPC module to your design. You will next need to add a PLL module to generate all clocks. Follow the same instructions as in ECE224 to generate clock c0 with 50Mhz, phase -3ns and clock c1 with 50Mhz, phase 0. Next add a third clock (c2) and configure it with 25Mhz, phase 0. Use c0 for the

DRAM_CLK, c1 for the input clk_0 to the SOPC block, and clock c2 for the input clk_1 to the SOPC block.

Finally, connect all pins. Note that all VGA pins are outputs, and all SRAM pins are outputs with the exception of DQ which is bidirectional. You can check the exact name of the PINS in the DE2_Pins file provided on learn (again, the same as in ECE224) - the assignment of outputs to pin names should be obvious. Note that compared to ECE224, the SD card interface has one additional bi-directional pin (SD_DAT3). Figure 5.2 shows our final schematic. Like in ECE224, we have connected the reset input to SW[16] (switch 16). Before you compile the hardware design, remember to assign pins using the DE2_Pins file, and to set all unassigned pins to tri-state under Device settings.

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. You can start with an empty project or a board test project (note that the board test project might try to drive peripherals which you have not included in your project). When you run the board, remember to check that switch 16 is set to high, otherwise your Nios II core will be kept in the reset state and the software tool will not be able to connect to it.

5.1 Video IP Cores

A detailed explanation of the Video IP Cores is provided in the video.pdf file available on learn. In particular, you should read Sections 2.2, 3.1, and above all the entirety of 4.10 (which includes function specifications and example code).

The VGA Controller 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 the VGA controller uses a resolution of 640x480 pixels and requires 30color bits per pixel, it effectively needs to be fed 640x480x30x60bits 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 Pixel Buffer DMA takes the frame information from a specified location in memory (the frame buffer or pixel buffer) and transfers them to the VGA controller.

There are, however, some caveats. First of all, the VGA Standard uses 10bits for each RGB color components, while it is more common to decode video in RGB format with 8bits per component, as explained in Chapter 4. Hence, a RGB resampler is used to change the pixel format. Second, to simplify the project we decided to allocate the frame buffer in a separate memory - SRAM instead of the SDRAM used as main memory by the Nios II. Since the SRAM is quite small, this limits the maximum size of the frame buffer that we can allocate. Hence, our project uses a frame buffer of size 280x200 pixels. **This means that we are restricted to decoding videos of size 280x200 maximum.** The Pixel

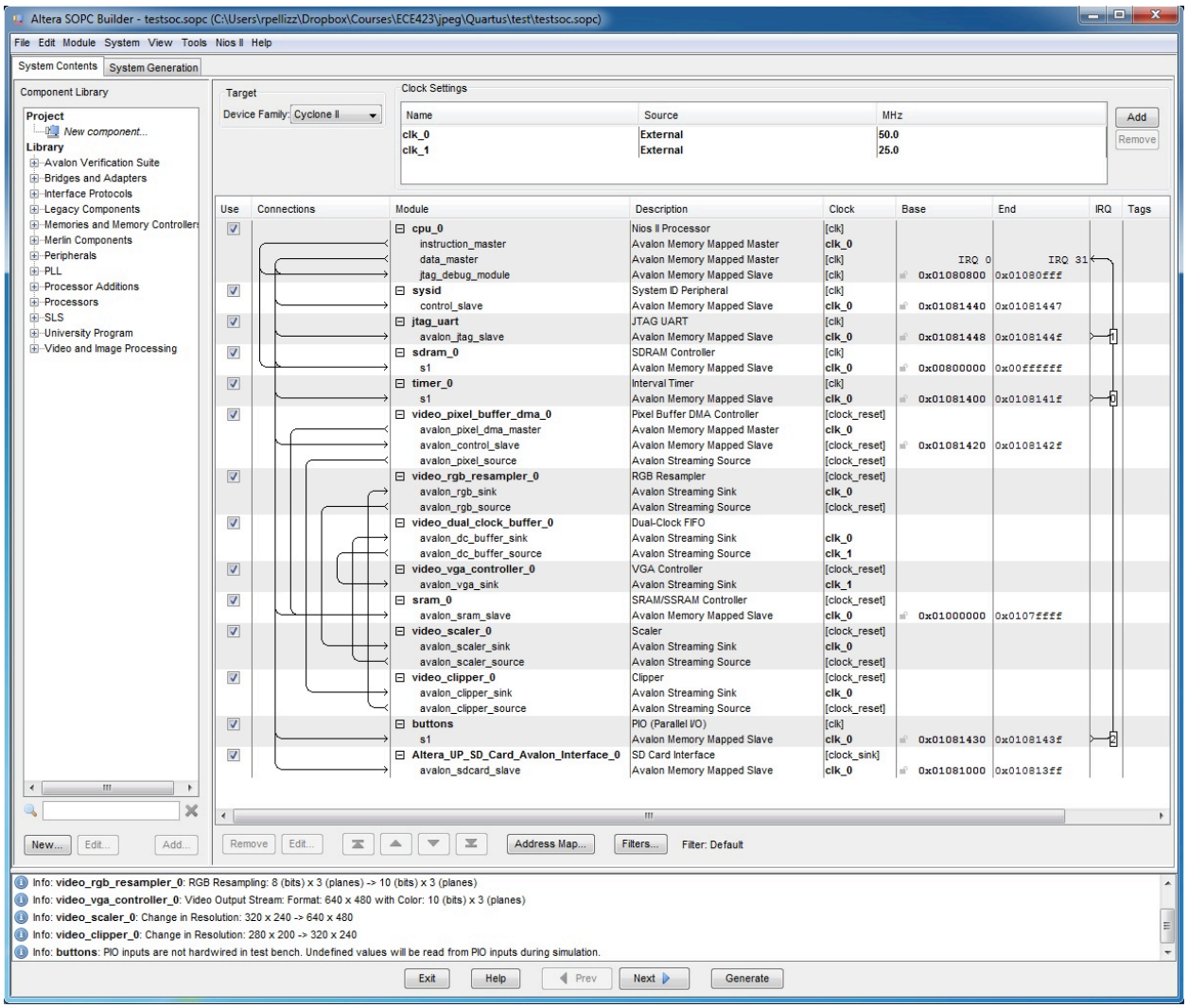

Figure 5.1: SOPC configuration

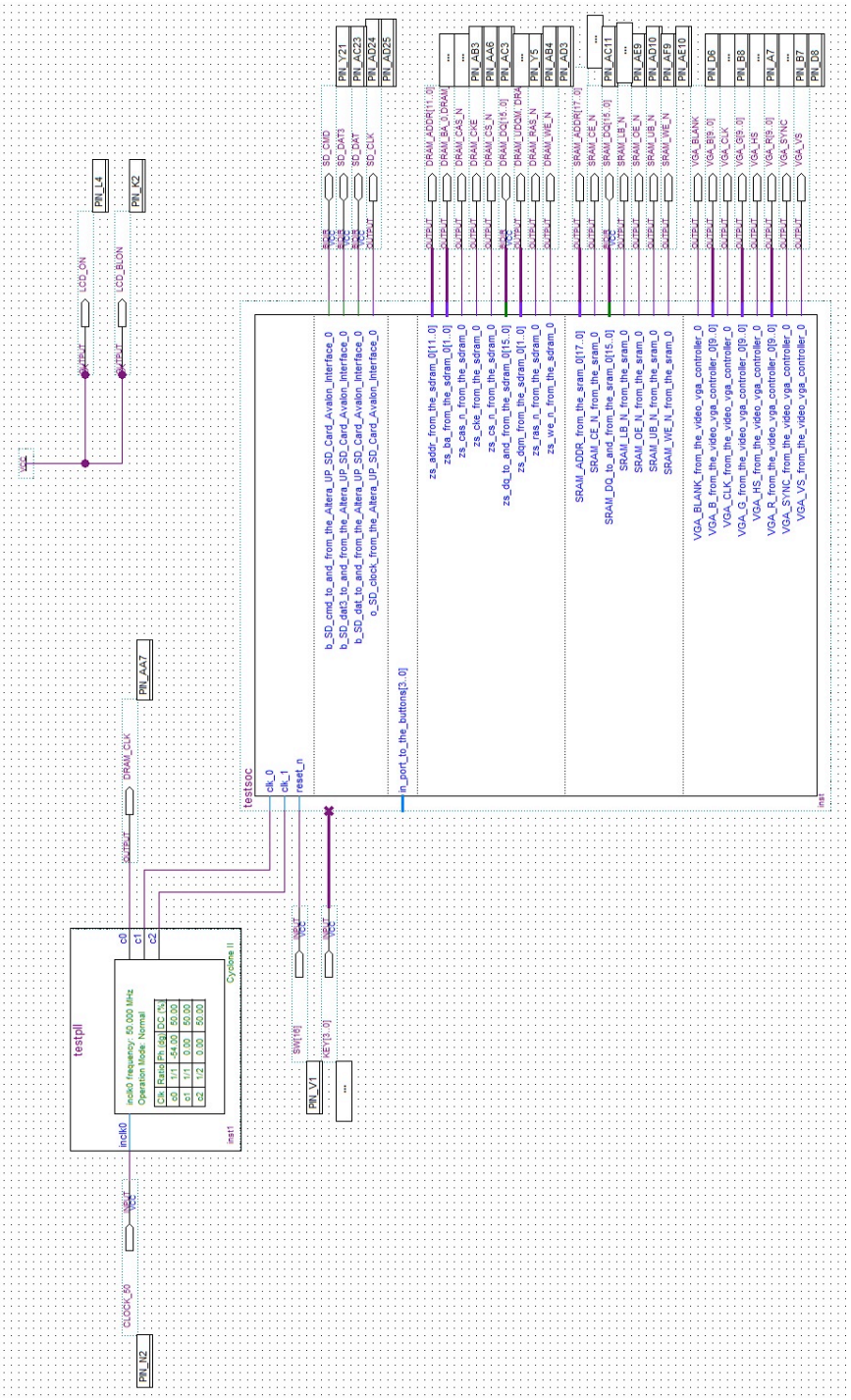

Figure 5.2: Quartus II bdf

Buffer DMA reads a 280x200 pixels, 24bits RGB buffer from the SRAM, and then Clipper and Scaler hw components are used to change the resolution to the 640x480 required by the VGA Controller. Notice that since each 24bits RGB pixel is effectively stored in a 32bit word (i.e., one byte is wasted), the size of the frame buffer is 280x200x4 = 224000 bytes.

A final important note is relative to using the frame buffer. Since the VGA Controller draws one line at the time, modifying the content of the frame buffer to show the next video frame while the VGA Controller 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 VGA Controller operate on one frame buffer (the front buffer), while the application updates a different buffer (the back buffer). Then after the VGA Controller finishes drawing the current frame, the front and back buffers are flipped.

The Altera HAL provides a set of useful functions to manipulate the Pixel Buffer DMA. Refer to Sections 4.10.4 and 4.10.5 in video.pdf for details. In general, to display a frame you should: 1. ensure that the Pixel Buffer DMA is not in the process of switching buffers (function alt_up_pixel_buffer_check_swap_buffers_status); 2. write the new frame information to the back buffer; 3. call the alt_up_pixel_buffer_check_swap_buffers function to swap the front and back buffer. You can draw individual pixels to the back buffer with the alt_up_pixel_buffer_draw function. However, this function (implemented in altera_up_avalon_pixel_buffer.c in the bps project) is extremely inefficient, since it performs several checks every time you write a pixel. Hence, you should rewrite the function by removing all checks (note that we configured the Pixel Buffer to use consecutive address mode and each pixel occupies a 32bits word as explained above). Before you get starting with porting the MJPEG423 decoder in Lab1, you probably want to write a simple test application that draws some pixels / boxes / lines out to the monitor just to test that the video chain is working.

5.2 SD Card Controller

A detailed explanation of the SD Card Controller Core is provided in the sd_card.pdf file available on learn. In particular, you might want to description of available HAL functions in Appendix B. Note that the provided HAL functions only allow you to read a file sequentially, one byte at a time. This is both highly inefficient, and insufficient to implement the project, since we need the ability to skip forward/backward in the video file. For this reasons, we implemented additional custom functions in the sd_card.h and sd_card.c files available on learn; please make sure to copy the files into your project.

The provided SD card uses the FAT16 filesystem. Under FAT16, each file is divided into a set of sectors that can be allocated anywhere on the SD card; each sector is 512 bytes long. Reading data from the SD card involves:

1. initializing the SD Card Interface;
2. opening the desired file;
3. obtaining the position of all sectors used by the file on the card by querying the filesystem;
4. determining the sector index and position within the sector of needed data. Assume that we wish to read a datum at position x within the file; then the required sector number is floor(x/512), and the position within the sector is x module 512.
5. Reading the sector from the SD card. The hardware of the SD card peripheral can be programmed to transfer the entire sector data into a 512 bytes hardware buffer implemented in the peripheral itself;
6. reading the desired data from the peripheral hardware buffer.

As an example of how to use the SD card peripheral, we provide sample code in file read_sd_example.c on learn. The file shows how to read the entire content on a file into an array in main memory. The following functions are used:

1. The HAL functions alt_up_sd_card_open_dev, alt_up_sd_card_open_dev, alt_up_sd_card_is_FAT16 should be called in sequence to initialize and test the peripheral.
2. The HAL function alt_up_sd_card_fopen can be used to open the desired file. The function returns a file_handle, which must be passed to further functions. The alt_up_sd_card_fclose is used to close a file.
3. The custom function sd_card_create_sectors_list must be called immediately after opening a file to obtain the sector information. **Note that you can only call the function on one file at a time; if you are trying to use multiple files, you must close the first file before opening the second one and calling sd_card_create_sectors_list again.**
4. The custom function sd_card_start_read_sector and sd_card_wait_read_sector are used to start a sector read operation and wait until the read operation is completed, respectively. Note that the wait function simply polls the SD Card Interface until the read operation completes.

Once a sector has been read by the interface, the sector's data can be obtained by directly accessing buffer_memory. Since the buffer is implemented in the peripheral itself, it should be read using IORD instructions.

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 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 need a timestamp counter (a timer). You should have added a timer in SOPC while creating the base system described in Chapter 5. However, if you add a single timer, by default the timer is configured in the BSP as the system clock timer, and cannot be used as a timestamp. To rectify this situation, in the Nios II Software Build Tool right click on your BSP project and go to Nios II, BSP Editor. The BSP editor screen should open on the main tab, settings, common, hal. Deselect your timer (ex: timer_0) from sys_clk_timer and select it under timestamp_counter. You will need to regenerate the BSP and rebuild the project.

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 50Mhz) 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 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 8.1; 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 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 Altera FPGA:

- Logic Elements: there are 33216 LE in the FPGA on the DE2 board.
- Memory bits: there are 430,080 bits of internal RAM, organized as 105 RAM blocks of 4Kbits (512 bytes) each.
- Embedded Multipliers: there are 70 9-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). Therefore, if you are trying to determine the resource usage of a single hw component, you should: 1. compile the design with the tested hw component; 2. compile the design without the tested hw component; 3. for each resource, take the difference between the resource usage reported by Quartus II for the two cases.

An important note on resource utilization. As you optimize your design in Lab3, you will likely want to use a significant portion of FPGA resources. It is generally ok to use up to 100% of the multipliers. You can also use 100% of the internal RAM as longs as you do not have internal fragmentation (the Cyclone II uses blocks of 4KBytes). However, you can not hope to get close to 100% in LE usage - as the number of used LE increases, the place and route steps has a harder and harder time trying to meet the timing constraints. Try to stay below 70-80% LE usage if possible.

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 based on the .map file generated by the Software Tools. The .map file for a given software application project can be found in the directory for that project (which is typically located under software in your Quartus II hardware project directory).

The .map file reports the allocation of all code and data used by the project in main memory. The various code and data sections are provided in order, starting from the .text (code) section and following with the data (read only, read write, initialized or non initialized); finally, the file might contain a set of comments and/or symbols. Your goal is to locate the last memory address allocated by the project; by subtracting from this value the address of the beginning of main memory, you can evaluate the size of the code + global data required by the application. Typically, the last section in the file should be the .bss section, followed by a .sdram_0 section that represents the beginning of the stack.

Finally, note that this method only provides the required size of code and global data; in practice, extra space is consumed by the stack and heap of the application (see Section 8.3.1 for further details).

Chapter 7

HW Coprocessors and Nios II

This chapter discusses how to create new hardware coprocessors and connect them to the Nios II as new custom instructions. All the material discussed in the chapter is taken from the custom_instruction.pdf document by Altera which is provided on learn; here we simply summarize the most relevant information.

7.1 Custom Instruction Overview

Custom instructions are directly connected to the pipeline of the Nios II processor. Since the Nios II uses a register file supporting two reads and one write per cycle, custom instructions are limited to 2 input operands (32 bits each) and 1 output result (also 32 bits). Figure 7.1 shows the block diagram of the hardware coprocessor for a custom instruction (note: some functionalities that are unneeded for the class project have been omitted from the diagram for simplicity). When you create a custom instructions, you have two main decisions regarding the behavior of the instruction: the instruction can either be combinatorial or sequential (multi-cycle), and it can be either simple or extended.

7.1.1 Combinatorial and Sequential (Multi-cycle) Instructions

A combinatorial instruction receives as input up to two operands (ports dataa and datab) and returns the result (port result) **within the same clock cycle**. A combinatorial instruction cannot use a clock or reset; it can only include combinatorial logic.

A sequential (multi-cycle) instruction uses a clock and can take multiple cycles to complete. All sequential instructions must use the clk, clk_en and reset signals. The optional start signal is used by the Nios II processor to inform your coprocessor that the

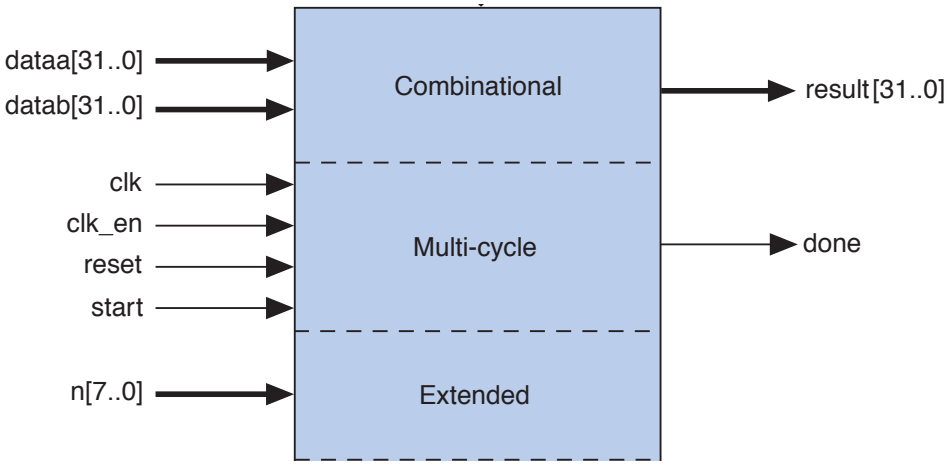

Figure 7.1: Custom Instruction Block Diagram

input operands are valid and that you should start processing the instruction. A sequential instruction can either be fixed length or variable length. In the case of a fixed length instruction, the processor expects the result to be valid k-th clock cycles after asserting start (where k is a configurable parameter determined at design time). In the case of a variable length instruction, your coprocessor must use the done signal to signal to the Nios II the completion of the instruction (meaning that result now holds valid data). A timing diagram for a multi-cycle variable length instruction is provided in Figure 7.2.

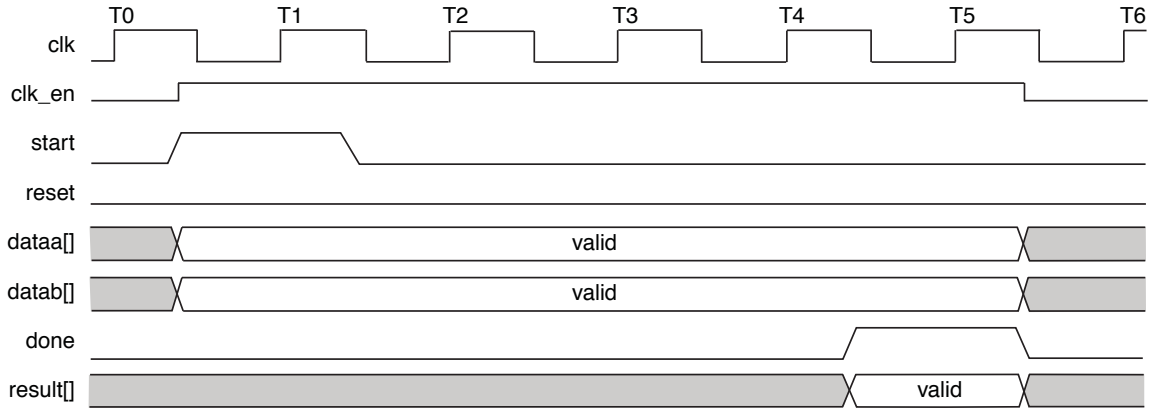

Figure 7.2: Variable Length Multi-Cycle Instruction Timing Diagram

7.1.2 Simple and Extended Instruction

Extended instructions implement different variations based on the value of an additional input port n; simple instructions do not include this port. Note that while Figure 7.1 shows a width of 8 bits for the n port, this is simply the maximum possible width; the width of n can be configured at design time to be any value between 1 and 8, depending on the number of required variations (i.e., if the width is 1, then there are two variations, for n = 0 and n = 1; if the width is 2, there are 4 variations, and so on).

You might want to use variations for different reasons:

- implement different instructions within the same hardware coprocessor. I.e., based on the value of n, you could perform different operations on the provided inputs.
- Implement coprocessors that require more then two inputs or one output: you can use variations to pass inputs from the Nios II to the coprocessor, two 32-bit registers every cycle, as discussed during lecture. You can then use other variations to get back multiple 32-bit results from the coprocessor. You need a clock to save values internally to the coprocessor, so in this case you should implement an extended multi-cycle instruction.

7.2 Implementing a Custom Instruction

To implement a new custom instruction, start by downloading the template file custom-intruction.vhd from learn; the file implements all ports required for a multi-cycle extended instruction. Save the file to your project directory and rename it with the name of your custom instruction. Similarly, edit the file to change the name of the included entity to the same name as the file. You should also edit the port declaration to remove and/or modify any unneeded port based on your design, as explained in Sections 7.1.1, 7.1.2.

You can then use the declared entity to implement your desired instruction in VHDL. If you want to create other entities using addition VHDL files and include them as components in your top-level entity, you can do so; just make sure to add all required VHDL files in Step 2 below.

Once your custom instruction is ready, perform the following steps to create a new SOPC component for your instruction:

1. Open SOPC builder, then click on "New component..." at the top of the Component Library on the left. This opens the Component Editor.

2. On the HDL Files tab, add your custom instruction VHDL file. The wizard will start processing your file. If no errors are found, the file will be added to the project list. The file should be listed as the "top" level file, and the "synth" setting should be turned on (this ensures that the file is compiled as part of Quartus II design synthesis); the "sim" setting is only needed if you want to generate a simulation model. The Top Level Module should indicate the name of the VHDL entity representing your custom instruction.

3. Move to the Signals tab. The wizard attempts to read the VHDL file you provide and automatically guess the correct signal interface for the device, but it is not always successful. A custom instruction uses the "nios_custom_instruction_slave" interface, so ensure that this is the interface listed for all signals, and that the Signal Type corresponds to the signal name (including width and direction).

4. Move to the Interfaces tab. If you have assigned all the signals as intended, the only interface with valid signals should be the Nios custom instruction slave. If the panel displays other interfaces, use the "Remove Interfaces With No Signals" button to remove them. Then, ensure that all interface parameters are correct (you might have to configure the number of cycles for a multi-cycle instruction with fixed length).

5. On the last tab (Library info), ensure that the name corresponds to the one of your custom instruction, then press finish and save your new SOPC component.

Once you have created the SOPC component, you can add the new custom instruction to your Nios II core:

1. In SOPC builder, click on your Nios II core to open the Nios II configuration wizard, then go to the Custom Instructions tab. The newly created instruction should appear in the list on the left. Simply use the add command to add the instruction to the Nios II ISA, then close the wizard.

2. Generate the SOPC system again by pressing the Generate button in SOPC, then once generation finishes, return to Quartus II, save the design, and start compilation. Note that if compilation fails due to errors in your VHDL file, you do not need to repeat this same procedure (unless you decide to change the ports in your top-level entity): you can simply correct your VHDL code and re-compile under Quartus.

3. Open the Nios II Software Build Tools and regenerate the BSP for your project, then open the system.h file in the BSP. The header should now include a new macro for your custom instruction. You can use the macro in your C code every time you want to invoke the custom instruction.

Note: if you created an extended instruction, note that the value of n is encoded in the opcode of the instruction itself. This means that the C compiler must be able to determine the value of n at compile time. Hence, you can only use constants for n in the C macro of the custom instruction.

Chapter 8

Multicore Design with Nios II

This sections explains how to create and code a multicore design using the Nios II and the Altera tools. Sections 8.1 and 8.2 discuss essential aspects for communication among processing elements, while Section 8.3 details how to create both the hardware and software design under Quartus II and the Nios II Software Development Tools.

8.1 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 two processing elements (ex: Nios II and the Pixel Buffer DMA, or two Nios II). 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. Even if the data is written back to main memory, a second 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 (in a multicore design, each Nios II has a private cache).

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 mech-

anisms to do so (additional details and function definitions are provided in the cache.pdf document on learn):

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).

8.2 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 component again implements a standard mailbox functionality with non-blocking send and either blocking or non-blocking receive.

To include a mutex or mailbox in your system, open SOPC builder and in the component library navigate to Peripherals, Multiprocessor Coordination. Notice that when you add a mailbox to your design, the wizard asks you to specify the memory component and location within that component to be used as a buffer for the mailbox messages. You have two options: 1. you can allocate a new memory component for the mailbox buffer. In this case, add a new on-chip memory RAM (Memories and Memory Controllers, On-Chip, On-Chip Memory), keeping into mind that you will consume an internal RAM block for every 4Kbits of buffer. 2. You can allocate the buffer in main memory (SDRAM). In this case, you must select an address range that does not overlap with the memory used for the code/data of any Nios II; see the next Section 8.3 for more details.

A complete description of the mutex and mailbox API can be found in the mutex.pdf and mailbox.pdf files available on learn.

8.3 Creating a Multicore Design in Quartus II

The facilities provided in the Altera SOPC builder and Software Tools make it reasonably easy to create a multicore design using the Nios II soft core. There are three main steps needed to obtain a functional design: 1. determine the memory map for the multicore system (Section 8.3.1); 2. add and configure additional Nios II cores (Section 8.3.2); 3. create the required application projects in the Software Tools (Section 8.3.3).

8.3.1 Multicore Memory Map

Since the DE2 board contains a unique main memory large enough to hold application code and data, the 8Mbytes SDRAM, all cores must execute out of the same memory. To ensure that the code and data of each core is allocated in disjointed memory addresses, the system must be properly configured. In a typical system, the most complex step in this process involves correct configuration of the linker on each core, but luckily, the Altera tools are able to automatically configure linker scripts for all cores based on the hardware configuration selected in the SOPC builder.

Figure 8.1 (taken from multiprocessor.pdf on learn) shows an example memory configuration for a set of six Nios II cores. Note that addresses in the figure represents offsets in the SDRAM main memory. The code entry point and exception address represent the address of the initialization code and of the interrupt vector table for a given core; you

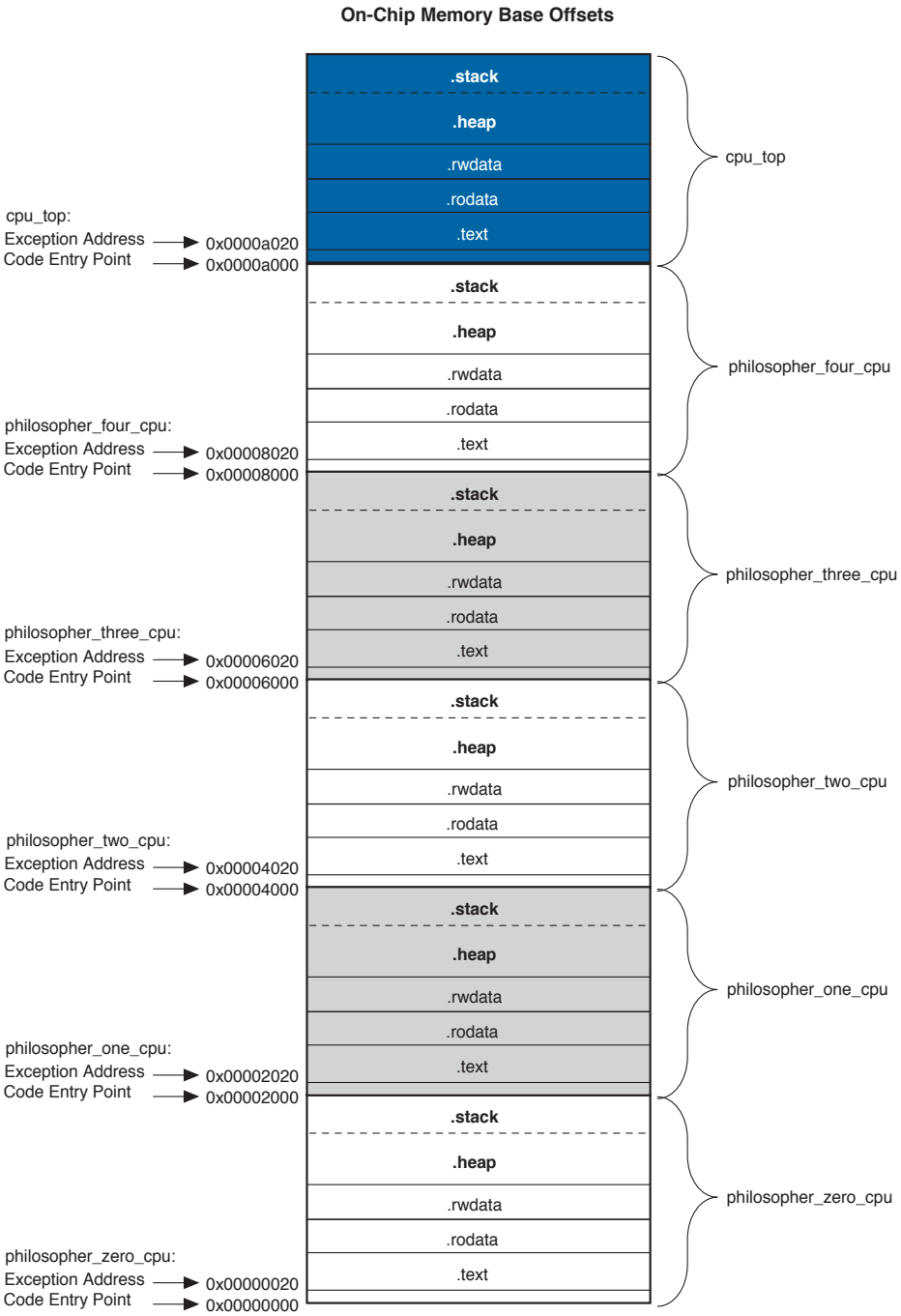

Figure 8.1: Multicore Memory Map

should always set the least significant byte of their addresses to 0x00 and 0x20, respectively. You can set the code entry point address and exception address for a Nios II core under SOPC builder by opening the wizard for that core and modifying the "reset vector" and "exception vector" addresses on the first tab.

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 (this is the case for the last core in Figure 8.1, cpu_top). As an example in the figure, each philosopher core uses a region of 0x2000 bytes, while cpu_top uses all remaining memory. Also as shown in the figure, within the memory region allocated to a core, the lower addresses are used for code (.text section) and global data (.rodata and .rwdata section); the rest of the memory region is used for the stack and heap, which grows in opposite directions - the stack from the top to the bottom, and the heap from the bottom to the top.

There are four important considerations when configuring the memory map. First, note that each core has a separate heap. 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 8.2 can then be used to pass the addresses of relevant buffers to other cores.

As a third consideration, note that the code entry point of the first core does not need to coincide with an offset of 0 (i.e., the beginning of main memory). If the code entry point offset is set to a value greater than 0, then the lower addresses in memory will not be used to allocate code or data of any core. These portion of main memory could be used to statically allocate other desired data structures, such as the mailbox buffers described in Section 8.2.

Finally, the memory region used by each core must be clearly sufficiently large to hold all code and data for the core itself. Note that failing to do so might cause an error either during software build (if the code + global data does not fit) or at run-time (if the code + global data fit in the region, but the remaining space for stack + heap is not large enough). The memory region size required to hold code and global data can be inferred using the technique described 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. Be sure to include some margins when configuring the system: if you have computed a required size of 80KBytes, using a memory region size for the core equal to the next power of two (128Kbytes) is likely a good idea.

8.3.2 SOPC Configuration

Follow these steps to add and configure additional Nios II cores to you project.

- Under SOPC builder, 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. You are free to configure the core type (s/f, with or without multiplier) and included custom instruction(s) as you see fit based on the results of your Design Assignment.
- 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. Under connections, ensure that the data_master of each core is connected only to the corresponding JTAG UART peripheral.
- Ensure that the data_master and instruction_master connections for the new core are connected to main memory SDRAM, then set the reset and exception vector on each core as discussed in Section 8.3.1. Connect the data_master to the sram used for the frame buffer only if the new core must be able to write to the back buffer.
- Connect the data_master of the new core to the sysid peripheral, as well as to any synchronization peripheral (mailboxes and mutexes) that the core needs to access.
- Outside of memories, sysid and synchronization, no other peripheral can be shared among multiple cores. Each other peripheral should be connected to only one core. In particular: each button, timer, SD card PIO, and pixel buffer dma 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 (IRQ line in SOPC main window) 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 timer for the new core, since timers cannot be shared), simply add it to the project and connect it to the right core as explained.

An example of a resulting dual-core project is shown in Figure 8.2; note that the design includes two mailboxes for communication between the cores.

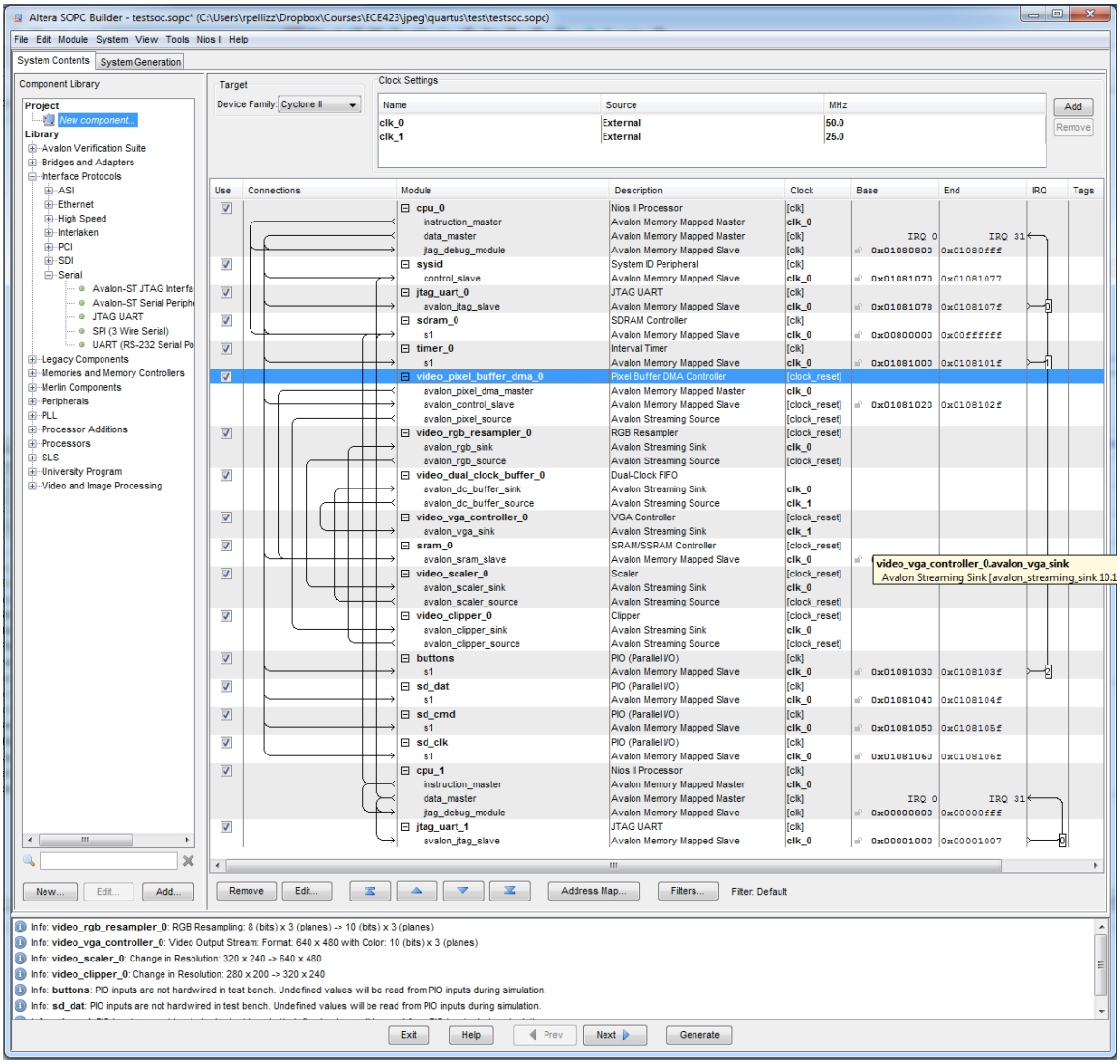

Figure 8.2: SOPC multicore configurations

8.3.3 Multicore Software Configurations

Unfortunately, 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. 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. No other configuration is needed; the Software Tools use the information from the SOPC builder to automatically configure memory regions for each core. 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.

To correctly start your multicore application, it is important to understand the precise behavior 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 8.3.2, 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 8.3.1), and to receive timer interrupts if needed.
- 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 DE2 system, 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 SOPC configuration.

Part III

Laboratory Projects and Assignment

Chapter 9

Lab1

9.1 Activities

Lab1 consists of the following activities:

9.1.1 Create the initial HW platform.

Following Chapter 5, create a custom Nios II system using Quartus II and SOPC builder which is able to read from the SD card and output to the VGA Controller.

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).

In particular, your application should perform as follows: 1. upon starting, find the file named "v3fps.mpg" on the SD card; 2. wait for the user to press pushbutton 0; 3. decode the file **as fast as possible**, sending each frame to the VGA controller as soon as it is decoded.

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 and it is the reason why we ask you to show the frame on the monitor as soon as they are decoded rather than periodically (based on the frame rate).

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

Note 5: you are only required to decode videos of size 280x200 pixels, since this is the size of the VGA frame buffer.

Note 6: the SD card is formatted according to the FAT16 filesystem and the video file is in the root directory. Remember to add the sd_card.c and sd_card.h files to your project as explained in Chapter 5.

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.

Note 8: make sure that you turn on all compiler optimizations for your code, or it will run even slower! You can do so by again going to Properties, Nios II Application Properties, and setting the Optimization Level to Level 3. You must do so separately for your application project and the bsp project, since they have a separate set of properties.

Note 9: the SD card contains various video files. We will be testing Lab1 with "v3fps.mpg", which is a small video designed to run at 3 frames per second (roughly the speed you can achieve on a single Nios II/f). This means that you can feasibly load the entire video file from the SD into memory (SDRAM) at the beginning of the application if you so desire, since the video file size if less than 2MBytes. However, please note that we will test Lab2 and 3 with a much bigger video that cannot entirely fit in main memory (the size of the SDRAM is 8Mbytes, and you must fit all code and data structures of the decoder as well), so you will have to load selected portions of the file into memory as you decode it. Try to come up with a solution that minimizes the amount of data copied in main memory (remember that the SD card peripheral copies the data from the SD card into an internal hardware buffer, not to main memory) to save memory bandwidth - this will be important for performance reasons in Lab 3.

Note 10: note that the RGB format used by the decoder (struct rgb_block_t) is the same as the format used by the VGA controller. Hence, you can optimize the video output step by allocating rgbblock at the same address as the backbuffer of the VGA controller

(in other words, in ycbcr_to_rgb you can directly write to the backbuffer in SRAM). Be careful that the backbuffer address is switched with the frontbuffer address every frame. Furthermore, since the VGA controller uses the Pixel Buffer DMA to read the frame information from the SRAM, you must handle cache coherency; make sure to read the relevant section in Chapter 8.

9.1.3 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 (either the entire file or each frame, based on how you implemented it); 2. lossless_decode; 3. idct; 4. ycbcr_to_rgb; 5. the code used to output the video to VGA, only if you did not implement it as part of ycbcr_to_rgb. For each block, report timing and memory utilization information.

For 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 Ybit-stream first, and then for the Cbbistream and Crbitstream. For idct and ycbcr_to_rgb, there is no significant difference between P and I-frames and Y or Cb/Cr blocks, so simply compute the time and memory utilization required to process all blocks in a frame.

First profile the application using the configuration described in Chapter 5 (using a NiosII/f with embedded multipliers). Then, change the Nios II settings, selecting none under hw multiply; and profile the application with this configuration (i.e., NiosII/f without hw multiply).

Note 1: write the profiling code once and test it on the original NiosII/f configuration. After you get the profiling to work, it is simply a matter of re-opening the Nios II system using SOPC and changing the core configuration, then compile the hw design and rerun your application. You might have to re-generate the BSP though.

If you finish early: you can get started with Lab2, or improve the application. In particular, note that Lab2 requires you to be able to pause/resume the video and to skip forward/backward some fixed amount of time by pressing corresponding push buttons.

9.2 Deliverables

Demo: a demo will be held at the end of Lab1b session. You must demonstrate the decoding application running on the Nios II, but no profiling is required at this point.

Report: write a max 3 page report detailing: 1. an overview of how you modified the sample application to execute on the Nios II, reading from the SD card and outputting

to the VGA; 2. a table reporting all required profiling information for the two processor configurations (NiosII/f with multiplier, NiosII/f without multiplier); 3. a short discussion of the contribution of each lab member. You must submit all your code (including the profiling code) together with your 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 Implement Playback Control

Improve your application by implementing playback control. The system should accept the following inputs: 1. pressing pushbutton 1 should cause the application to cycle among the three .mpg files provided on the SD card; 2. pressing pushbutton 0 should start and then pause/resume the video; 3. pressing pushbutton 2/3 should cause the video to skip behind/ahead by **roughly** ten seconds.

Note 1: as discussed in Lab 1, one of the .mpg files is too large to completely fit into main memory. Hence, you will need to load each frame as you decode it.

Note 2: the sample MJPEG423 decoder reads the footer of the file upon starting. As discussed in Chapter 4, the footer 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 10 seconds, simply look for an I-frame that is roughly 240 frames ahead/behind the current frame and restart decoding from that frame.

10.1.2 Create a Coprocessor for Y'CbCr to RGB color conversion.

A file named ycbcr_to_rgb.vhd is provided on learn. The file implements a combinatorial custom instruction for the Nios II in VHDL that performs Y'CbCr to RGB color conversion according to the MJPEG423 specification and the following interface, where dataa and datab are the 32-bits input registers and result is the 32-bit output register for the instruction:

- **Inputs:** the custom instruction expects the Y value to be passed in bits 7-0 (least significant bits) of dataa. The Y value must be in the range $[-128, 128]$ and encoded in two's complement. The Cb value must be passed in bits 7-0 of datab, and the Cr value in bits 15-8 of datab. Both must also be in the range $[-128, 127]$ in two's complement.
- **Ouputs:** the custom instruction returns blue component in bits 7-0 of result, green component in bits 15-8, and red component in bits 24-16. The RGB components are in the range $[0, 255]$.

Following Chapter 7, create a custom Nios II instruction under SOPC using the provided ycbcr_to_rgb.vhd file, and attach the custom instruction to the NiosII/f processor used in your sequential design in Lab1. Then, use the new custom instruction to speed up the execution of the color conversion function (also named ycbcr_to_rgb) in the MJPEG423 decoder application.

10.1.3 Create a Coprocessor for IDCT

Finally, create a custom instruction to speed up the execution of the idct function. To help you in this endeavor, we provide on learn a VHDL module (idct1D.vhd) which performs the 1D IDCT (see Chapter 4 for details on how the 2D IDCT is computed based on the 1D IDCT).

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).

Note 2: We strongly suggest that you start by implementing a custom instruction that performs a single 1D IDCT every time it is invoked. However, for full points you should use the provided code to implement a custom instruction that performs the whole 2D IDCT for one block.

Note 3: Since the provided module takes 128 bits as input and returns 128 bits as output, you will not be able to pass and return all arguments in one clock cycle. This means that you will need to implement an extended multi-cycle instruction - you can use the clock provided to multi-cycle instructions and the index provided to extended instructions to store data passed by the CPU over multiple cycles in internal registers in the custom instruction, and then use those to provide inputs to idct_1D.vhd (similarly for outputs). You should carefully think how to optimize argument passing given these constraints, especially when you implement the whole 2D IDCT.

10.1.4 Profile the Application and the Platform

Profile again the application in the same way as in Lab1, for the same three Nios II configurations, but this time after adding the two described custom instructions. You must report the new time for the two functions that you improved using custom instructions, i.e., ycbcr_to_rgb and idct.

Then, using the technique described in Section 6.3, derive the FPGA resource utilization for the following components: 1) a Nios II processor in each of the two configurations; 2) your designed custom instructions.

Note 1: when you are estimating the FPGA resource utilization of a Nios II processor, you should add a second Nios II to SOPC builder rather than removing the one you have to avoid issues when compiling the SOPC design.

If you finish early: you can start working on the Design Assignment. If you want to further optimize the design in hardware, note that there are other functions that are good targets for hardware implementation (for example, input_AC in lossless_decode.c).

10.2 Deliverables

Demo: a demo will be held at the end of Lab2b session. You must demonstrate playback control, the ability to play all three mpg files, and the use of the custom instructions, but again no profiling is needed at this point.

Report: write a max 4 pages report detailing: 1. the design of the playback control; 2. the design of the custom instruction for IDCT; 3. a table reporting all required profiling information and FPGA utilization; 4. a short discussion of the contribution of each lab member. You must 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%
Playback control design	20%
IDCT design	30%
Profiling Information	20%
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, as well as their configuration (Nios II version, custom instructions, etc.).
- Mapping and scheduling: determine which functional blocks are executed on which Nios II core, 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.
- 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, embedded multipliers, internal RAM, as well as space in the SDRAM main memory), as whether you can feasibly execute the application based on the chosen schedule (i.e., can you finish decoding each frame in time?).

While not required, you might consider designing two systems:

- a dual core system, with the objective of running the application at the maximum possible frame rate. While it is unlikely that you will be able to reach 24 frame per seconds with just two cores, for Lab 3 we require that you demonstrate the application running on a dual-core system. However, this is not a requirement - based on your design, it might actually be easier to run the parallelized application on 4 cores, for example.

- A general multi core system, with the objective of running the application at the full 24 frames per seconds. If your design can reach the target of 24 fps, then you should optimize it to minimize the utilization of FPGA logic elements. You must provide this more general design as part of the Lab 3 report (which is due after the demo).

The remaining sections in this chapter provide important information on how to perform the Design Assignment.

11.1 Application Scheduling

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

As an example, Figure 11.1 shows an example schedule derived in class, with functional blocks representing the SD read operation (R SD), Lossless Decode on Y/Cb/Cr components (LD), IDCT on Y/Cb/Cr components (ID), and finally Color Conversion (the ycbcr_to_rgb function, CC in the figure). In the figure, the schedule is repeated every 10 time units, while the makespan of the graph (time required to process an entire frame) is 20 time units. Note that the figure makes some simplifying assumptions, in particular, that the time for each functional block is independent of the type of frame (I or P-frame); in practice, the time for I and P-frames can vary widely, hence you should account for it similarly to what discussed in class. Note that for the videos used in the lab, the minimum separation between successive I-frame is 9 frames. Also note that while in the example we used a single block for the entire lossless decoding, IDCT processing on each color component and the entire color conversion operation, the functional blocks could be broken down at the level of individual 8x8 blocks as discussed in Section 4.8.

The advantage of computing a static schedule for a task graph is that it greatly simplifies implementation. In the schedule in the figure, the same 5 operation are repeated every period (frame) on the first Processing Element: read from SD, LD Y, LD Cb, LD Cr, and Color Conversion. Hence, the application executed on the first core can simply comprise a loop executed periodically that call the 5 function calls in a fixed sequence.

11.1.1 Buffering

If the makespan of your application is larger than the period, than you must carefully account for buffer size, as discussed in class. As an example in Figure 11.1, note that the

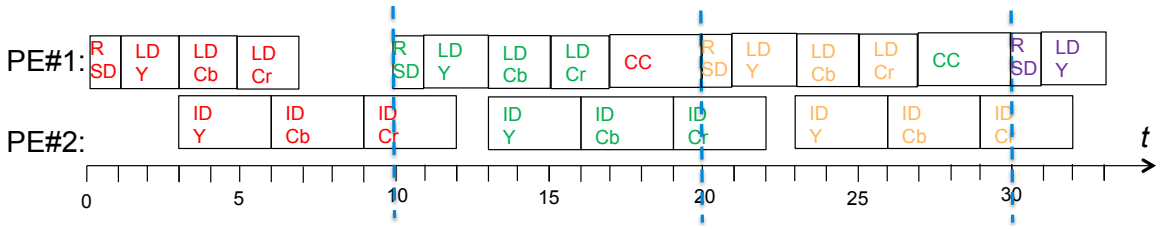

Figure 11.1: Example Task Graph Schedule

CC block for the first frame (red) is executed after the IDCT Y block for the second frame (green). Hence, you will need to allocate two copies of the Yblock data structure (i.e., a buffer size of two) to avoid overwriting the data: odd frames can use the first copy, while even frames can use the second copy. The same strategy must be used for each other data structure that can be overwritten; note that if the makespan is larger than two periods, you might need a buffer size of 3 or larger.

Pay special attention to the frame buffer; by construction, the frame buffer must be of size 2, since we only have space to allocate a front buffer and a back buffer in the on-board SRAM. Assuming that the ycbcr_to_rgb function writes directly to the back buffer, this means that you must be careful not to execute the Color Conversion block more than once every period - otherwise, you could overwrite the content of the back buffer before you are able to switch the back and front buffer with the alt_up_pixel_buffer_dma_swap_buffers call.

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; as an example in Figure 11.1, the ID Y block cannot be executed on the second core before the LD Y block finishes executing on the first core. 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 on which you are executing the first functional block in the task graph - likely the read from SD card function. You can use a periodic timer on the master core to start the periodic execution by running the first functional block, followed by the the other blocks in the fixed periodic sequence for that core (see the example in Figure 11.2).
- Use a mailbox to activate each task that depends on a task running on another core. This behavior is exemplified in Figure 11.2 for the second frame (green) in the same example as in Figure 11.2. Before executing the ID Y block, the second core should wait on a mailbox. Conversely, the first core can write to the same mailbox after

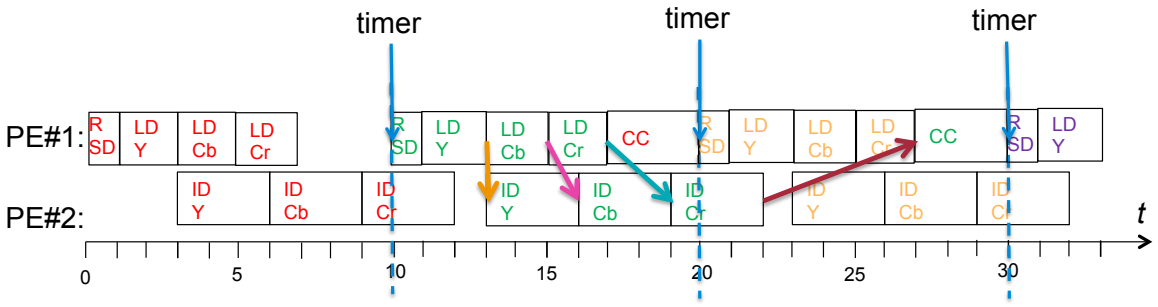

Figure 11.2: Example Synchronization

completing the execution of the LD Y block; this ensure that ID Y can only run after LD Y completes on the other core. The mailbox message can conveniently contain the address of the data structure (buffer) that ID Y needs to operate on. The same strategy can be used between LD Cb and ID Cb and between LD Cr and ID Cr (in fact, the three pairs of blocks could use the same mailbox, since they are processes in sequence). Another mailbox must be used to synchronize between the end of the ID Cr block and the start of the CC block for the same frame.

- A final note is relative to the pixel buffer dma. To ensure that you display video in a smooth manner, the frame buffer should we swapped periodically in a consistent manner. We suggest employing the same periodic timer used on the master core to start the periodic processing to also swap buffers. In other words, every time a periodic timer is received on the master core, the application can first call the alt_up_pixel_buffer_dma_swap_buffers, ensuring that the last processed frame is displayed, and the start the execution for the next frame.

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 that you can finish running each periodic function sequence within the allocated period.

The first main source of overhead is due to synchronization operations, i.e., reading and writing to mailboxes. If you only access mailboxes a small number of times per period as in Figure 11.2, you can likely safely ignore this overhead. However, if you decide to synchronize the application at the level of 8x8 pixel blocks, the number of mailbox 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; 2. add the time required to manipulate mailboxes to the maximum running time of each functional block; 3. compute the application schedule based on such inflated time.

The second, and more significant source of overhead is memory contention. Since all cores share main memory SDRAM, concurrent accesses to the main memory are significantly slowed down, as discussed during lecture. You must either account for or mitigate this effect, otherwise your implementation will run much slower than what predicted based on the profiling in Lab1 and 2. This is an overview of the suggested procedure:

- Based on the profiling information from Lab2, determine which functional blocks are more memory intensive (i.e., the have high memory bandwidth, computed as size of data read/written in memory / execution time of the function). Try to avoid scheduling two or more memory intensive blocks at the same time on different cores.
- Scheduling alone is likely to be insufficient to prevent significant memory contention. You should consider reducing the load in main memory by allocating scratchpad memory to hold data buffers. You can allocate a scratchpad by adding a new on-chip memory RAM (Memories and Memory Controllers, On-Chip, On-Chip Memory) in SOPC builder, similarly to what is described in Section 8.2; each on-chip memory RAM block can be accessed in parallel, thus increasing the amount of available communication bandwidth. For example, in the sample application, the DCAC buffers outputted by lossless_decode are allocated to main memory; in this case, both lossless_decode and idct must access main memory to write/read the buffer content. If the DCAC buffers are allocated in a scratchpad, a core executing the lossless_decode function would be able to access the buffers while, for example, another core executing the ycbcr_to_rgb function accesses color buffers in main memory without suffering interference. However, keep into mind that the amount of on-chip RAM is quite limited; hence, you will have to think carefully how to execute the application and allocate the buffers. In particular, note that synchronizing the application at the level of 8x8 pixel blocks might significantly reduce required buffer size, hence allowing you to fit more buffers in scratchpad, but it increases synchronization overhead as previously noted.
- A certain amount of memory contention will likely be inevitable. After you optimize both the schedule and the buffer allocation, you can use the techniques described in class to determine bounds on the overhead suffered by the various functions.

11.2 Platform Configuration

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. A set of application-specific recommendations are provided in Section 4.8. Here we provide some other general remarks.

- Execute each functional block on the most suitable Nios II configuration: both idct and ycbcr_to_rgb can be significantly sped up by using custom instructions. Hence, you should configure the core onto which these functional blocks execute with the corresponding custom instruction.
- Do not use a feature if it is not needed: if a functional block has no speed up from running on a core with embedded multiplier, you should consider turning the multiplier off to save area.
- Do not overload a core: 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 probably not take longer than 37-38ms per frame. The only possible exceptions are I-frame; 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. However, beware that this complicates the schedule (as discussed in class).
- Execute the application as a pipeline: you do not need to complete execution of the current frame before starting processing the next frame; you can assign functional blocks to different processors and execute those blocks in a pipeline fashion. Just be careful about required buffer sizes.
- Add margins to your design: we discussed overheads in terms of timing, but make sure that your design also fits in terms of FPGA resources and main memory size. Do not try to reach 99% resource utilization - a simple misevaluation could result in a non-working design.

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 SOPC builder. While it is not required, so probably want to start with a simpler dual-core system and familiarize yourself with the procedure required to run multicore applications by running two concurrent applications. Then, create your demo platform based on the results of your Design Assignment.

12.1.2 Execute the parallel MJPEG423 decoder

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

Note: as part of the Design Assignment, you must make certain assumptions on the execution time of the various functional blocks; in particular, you must be able to complete the execution of a periodic sequence of functional calls on each core within the allotted time for a frame. Since the multicore platform will introduce additional timing overhead as explained in Section 11.1.3, you should ensure that when you run the application you can indeed meet the timing constraints. Our suggestion is to simply halt the application if you determine that one periodic execution takes longer than expected, i.e., it is not completed by the time the next timer interrupt arrives.

12.2 Deliverables

Demo: a demo will be held at the end of Lab3b session. You must demonstrate the decoding application running on **at least a dual-core** system. Note that for demo we do not grade your performance in terms of frame rate; however, we assume that the design is reasonably optimized (i.e. the schedule should be reasonable based on your chosen number of cores) and that you can display the video with a consistent frame rate, even if the frame rate is lower 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 (number and types of cores; mapping of functional blocks onto cores); B. application scheduling and synchronization, including required buffer sizes and buffer allocation; C. feasibility analysis (ensuring that you have enough resources to implement the hw/sw system; this should include an estimation of timing overheads). 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. You must submit all your code together with your report. The mark breakdown percentage for the report is reported in Table 12.1

Note 1: if you decide to prepare a dual-core system for the demo and cannot meet the 24 fps requirement, you should include two designs in your final report: 1. the design of the dual-core system that you demoed; 2. the design of an optimized system (with more cores) that can hopefully meet the frame rate requirement.

Note 2: as shown in Chapter 1, Lab 3 includes an additional grading component based on the performance of your implemented system. Our evaluation is based on the **implementation discussed in the final report**. Hence, if you want you can take the time between the Lab 3 demo and the report due date to improve your implementation. If the performance between Lab 3 and the report is significantly different, you should schedule a short meeting with the instructor to demonstrate the new implementation.

Total Lab 3 Report	12%
Written Mechanics, Organization, Clarity	20%
Design: Platform Configuration and Mapping	15%
Design: Scheduling and Synchronization	25%
Design: Feasibility Analysis	15%
Implementation and Results	20%
Group Member Contribution	5%

Table 12.1: Mark Breakdown For Lab 3 Report