ECE423 Embedded Computer Systems

Lab Manual

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, 2023

Contents

List of Tables

List of Figures

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 for the MJPEG423 format on a Xilinx Pynq Z1 FPGA board. 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 presents three main challenges:

- Parallelism: the MJPEG423 decoder is computationally intensive; you will not be able to run it at a satisfactory speed on a single processor core (ARM A9). Therefore, you will need to run the application in parallel on multiple Processing Elements (PEs).
- HW accelerators: some of the functional blocks in the application can be more efficiently realized as custom hw components in VHDL rather than running as software modules on a general purpose ARM A9 core. Hence, you will design an hw accelerator that can be run together with the software parts of the applications executed on the core(s). Rather than coding a complex VHDL modules like in ECE327, you will generate the accelerator starting from a C software description using a High-Level Synthesis (HLS) flow.
- 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 accelerators 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.

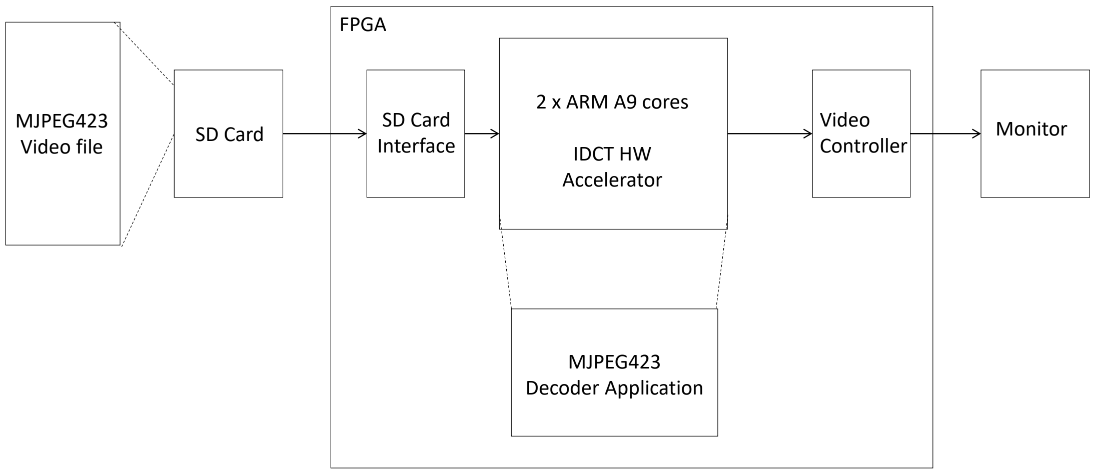


Figure 1.1: Decoder System Block Diagram

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 PEs: ARM A9 processors and hardware accelerator) 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 video IP blocks. Your objective is to ensure that you can meet frame rate requirements of the application; or assuming you cannot meet the frame rate requirement, simply maximize the frame rate.

The rest of the manual is organized as follows. Chapter 2 discusses how to form groups, deadlines, and other relevant lab policies. Chapter 3 provides the specification of the MJPEG423 format; you should familiarize yourself with the basic operation of the standard before starting on the lab. Finally, Chapters 4-5 detail the various lab activities and deliverables. (The description for Assignment 3 will be added when it is finalized - around the middle of term.) An overview of the activities for each assignment and a timeline for the completion of the project are further discussed in the next section.

1.1 Project Overview and Timeline

There are three assignments required to complete the project. Some of the class tutorials will be used to introduce the various assignments. Each assignment builds on top of the previous one.

In Assignment 1, a basic SoC system will be provided to you to execute the sequential MJPEG423 decoder. We provide the code for the decoder as well, but to execute it on the FPGA board, you will need to interface it with the SD and video 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 Assignment 2, you will create a hardware accelerator to speed-up the execution of the most critical part of the application - the Inverse Discrete Cosine Transform (IDCT). You will also profile the performance of the hardware components.

In Assignment 3, you will use the performance characterization derived in Assignments 1 and 2 to produce an actual design for the system, that is, specify: 1. the mapping of functional (software) blocks to PEs; 2. the scheduling (that is, the temporal order) of such functional blocks.

The required assignment deliverables are described in details in Chapters 4-5. Deadlines and grade breakdown are posted on the course outline.

All course project activities are to be performed in teams of 2 students. Details on how to form groups are provided in Chapter 2.

Chapter 2

Lab Policies

Lab Groups All lab activities are performed in groups of 2 students. You are free to select your partner. A sufficient number of groups for all students in the course have been created in Learn. You can form your group by simply joining one of the groups on Learn. **Make sure you form your group by Wednesday Jan 17 at 11:59PM.**

Note: if the number of students in the course is not even, we will form one group with 3 students.

Lab Access The lab room is E2 2356A. The lab is equipped with the required software, boards, cables and SD cards. After-hour access to the lab is available; the access code is 969421. Do not share the access code outside of this class.

Help sessions will be scheduled on the following days of the week and times, in the weeks prior to assignment deadlines:

- Mon 2:30PM-4:00PM
- Weds 10:00AM-11:30AM.

For assignment 3, each group will book a time to meet with the course instructor during the week of Mar 18 - Mar 22 (this is not mandatory, but encouraged to check that the devised schedule is feasible for implementation).

Deliverables Each assignment's deliverables consist of a demo and a report submission. Each group needs to book a time to meet with one of the TAs and perform the demo during the scheduled demo weeks. The link to a sign-up spreadsheet will be provided on Piazza once it's ready. We expect both group members to attend each demo; if this is impossible due to motivated reasons such as illness, please contact the Lab and Course Instructor. The report must be submitted through Learn; there are separate dropboxes for each assignment. In addition, you are required to submit the code used in each demo immediately after the demo itself to the same dropbox (see instructions in each assignment deliverable).

Late Policy You are granted a cumulative total of 6 grace days for late report submission over the term. This means that you can, for example, submit the first report 4 days late, the second report 2 days late, and all other reports by the deadline, and not be penalized. After consuming the 6 grace days, late reports will be graded as 0. 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.

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

Looking at, reviewing, or referencing code or solutions from versions of this course in previous years is prohibited. This includes reviewing solutions from previous students of this course.

Chapter 3

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; specifically, you should understand the key processing steps of entropy decoding, dequantization and IDCT described in Section 4 of the paper. 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. The remaining sections in this chapter will detail the MJPEG423 file format and the differences between the MJPEG423 decoder and the JPEG baseline.

3.1 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.

3.2 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 3.1 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 3.1: Y'CBCR conversion

Since each color component is encoded with 8 bits, a RGB color pixel is represented by 24 bits (3 bytes). In bmp files, each pixel is instead represented in the alternative RGBA format on 32 bits (4 bytes): the additional A(lpha) color component determines the transparency of the pixel, which we set to 0. Also note that the equations in Figure 3.1 are expressed in terms of floating point multiplications. However, for compatibility with processors that do not support a floating point unit, the sample decoder implements the conversion using fixed-point arithmetic in the *ycbcr_to_rgb* function (read the function code for details); note that the function accepts as inputs one Y' component 8x8 block, one CB component 8x8 block and one CR component 8x8 block and outputs 8 lines of 8 pixels in RGBA format.

3.3 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 3.2, 3.3: 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); these 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 $2^{\mathrm{CONST\_BITS}}$ (where $\mathrm{CONST\_BITS} = 13$), 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 $2^{\mathrm{PASS1\_BITS}} = 4$; 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 $\mathrm{CONST\_BITS} - \mathrm{PASS1\_BITS}$ 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 $\mathrm{CONST\_BITS} + \mathrm{PASS1\_BITS} + 3$ 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.

3.4 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 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

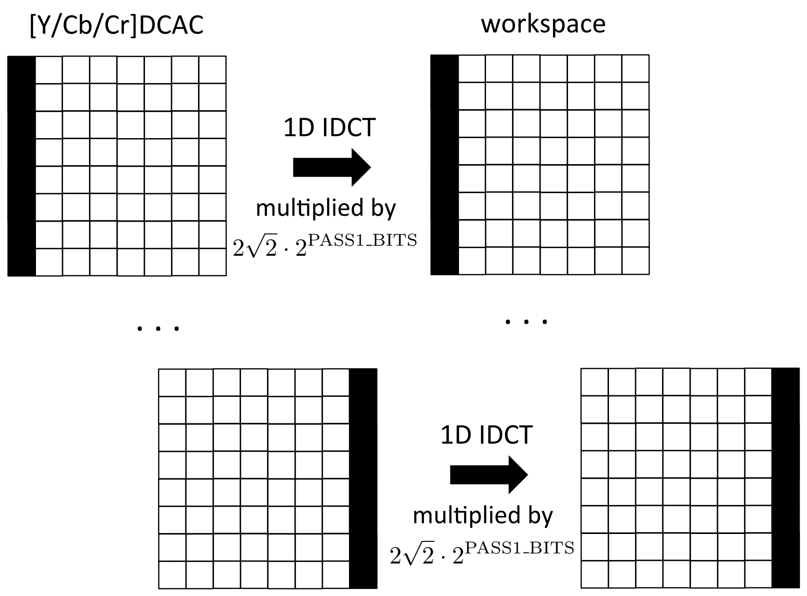

[Y/Cb/Cr]DCAC
workspace
1D IDCT
multiplied by
$2\sqrt{2} \cdot 2^{\text{PASS1\_BITS}}$
1D IDCT
multiplied by
$2\sqrt{2} \cdot 2^{\text{PASS1\_BITS}}$

Figure 3.2: 2D IDCT Pass 1: Column Processing

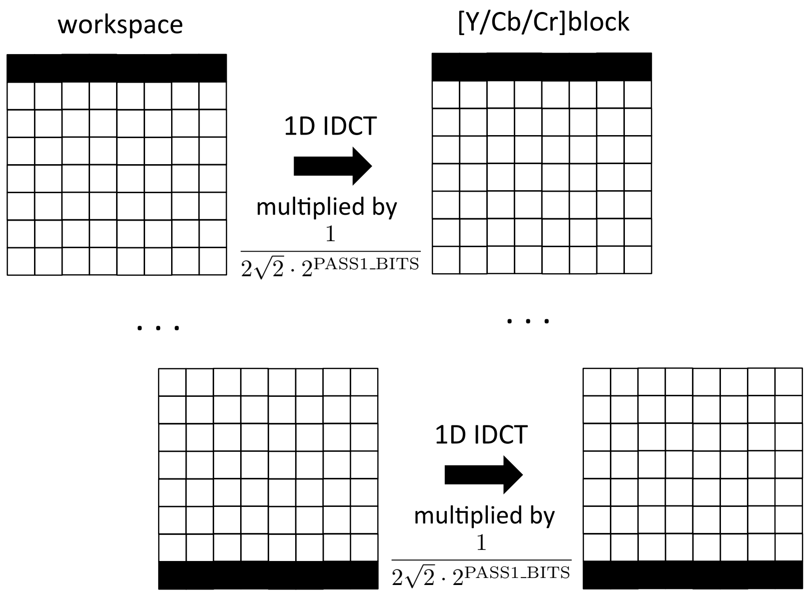

workspace
[Y/Cb/Cr]block
1D IDCT
multiplied by
$\frac{1}{2\sqrt{2} \cdot 2^{\text{PASS1\_BITS}}}$
1D IDCT
multiplied by
$\frac{1}{2\sqrt{2} \cdot 2^{\text{PASS1\_BITS}}}$

Figure 3.3: 2D IDCT Pass 2: Row Processing

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.

3.5 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).

3.6 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 (24 frames, equivalent to one second, for the example videos provided on learn).

3.7 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 5 4-bytes fields, as follows:

1. uint32 num_frames: the total number of frames in the video.
2. uint32 w_size: the width of the video in number of pixels. This value must be a multiple of 16.
3. uint32 h_size: the height of the video in number of pixels. This value must be a multiple of 16.
4. uint32 num_iframes: the total number of i-frames in the video.
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, Intel, ARM and RISC-V architectures are all little-endian.

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

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

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

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

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

3.8 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 ARM A9 core. Hence, your goal by the end of Assignment 4 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 3.4, 3.5 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 RGBA pixels in the rgbblock matrix (see also Section 3.2).

When you produce a design for the parallel decoder in Assignment 3, you will need to perform mapping, i.e., decide how to assign the various functional blocks to Processing Elements (PEs - ARM cores and HW accelerator). Each node in the DAG (i.e, an instance of a functional block executed on one image block) should execute on one PE, but apart from this limitation, you are free to explore different mappings. In particular, different nodes of the same functional blocks could be executed either on the same PE or on different PE (ex: you could split the lossless_decode nodes on 2 ARM cores). This 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

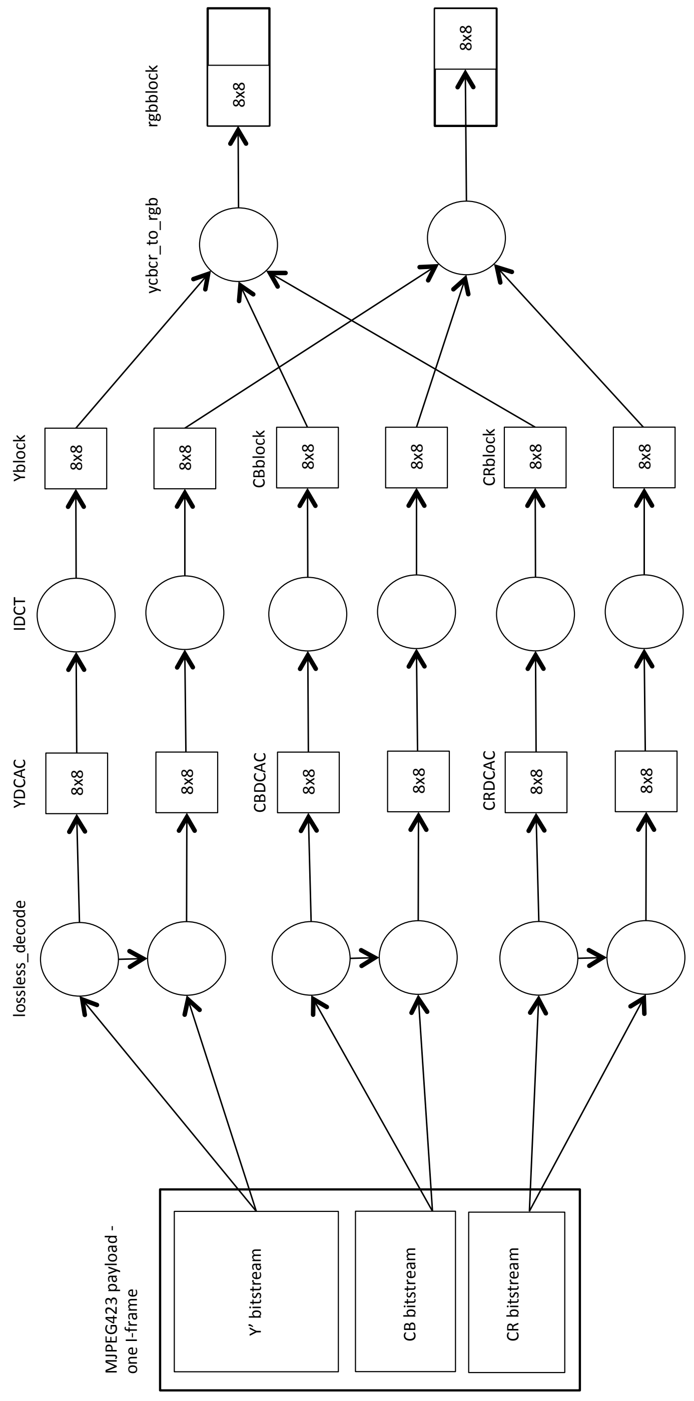


Figure 3.4: I-Frame Processing DAG

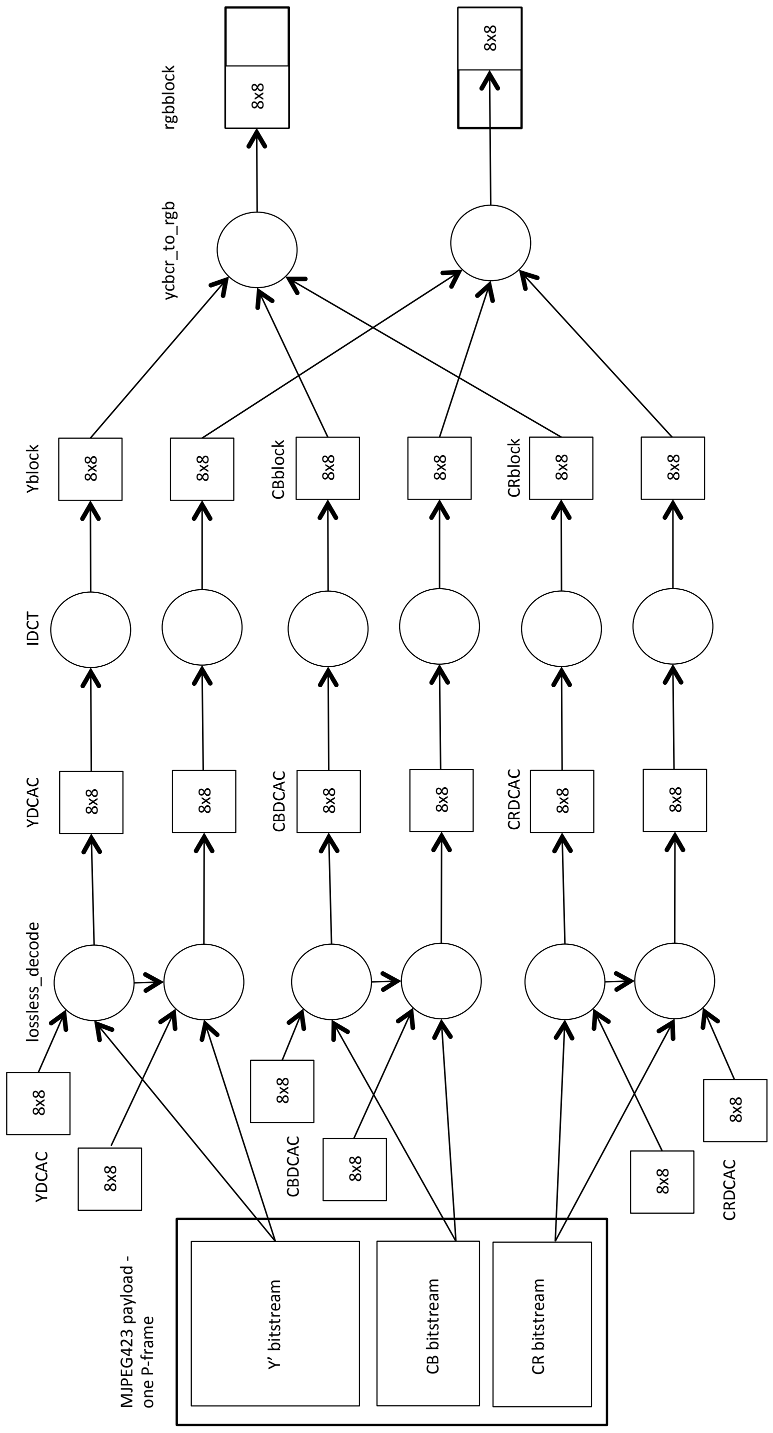


Figure 3.5: P-Frame Processing DAG

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

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

3. Finally, notice that each block can be processed in a pipeline. In other words, you do not need to wait for each of the first two stages (lossless_decode and IDCT) to finish processing all their 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.

3.9 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 pynq board in Assignment 1, 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 3.4, 3.5). 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 4

Assignment 1

In Assignment 1, you will port the sequential MJPEG423 decoder application to run on top of a provided system-on-a-chip design, implemented on a Pynq Z1 FPGA board. Section 4.2 discusses how to configure the board with the provided initial hardware design for Assignment 1, while Section 4.5 shows how to run a single-core application on one of the available ARM A9 hard-cores. Sections 4.6, 4.7, and 4.8 then detail the main tasks that you need to complete in this lab, and Section 4.9 summarizes the required deliverables.

4.1 Vivado Project Setup

Please follow the steps below to initialize the full Vivado project that we will be using for all the assignments:

- ☐ Start by downloading the `ece423_prefab.zip` file from Learn.

As is common in most development environments, a Vivado project has its own working directory. For Vivado, the working directory should be on a local drive (i.e. C: drive). A network or cloud drive should not be used as network bandwidth limitations severely compromise Vivado build performance.

- ☐ Unzip the prefab into a directory of your choice on the local drive (e.g. `C:/Users/<user_id>/ECE423/`). No spaces are allowed in the project path as it might create compilation errors. **Avoid using a network drive, like N: drive, as your working directory, as this will break the tools.**

Note that after you log out from a lab PC, the content of the users directory (or any other local directory) might be wiped. For this reason, ensure that you copy the whole project to external storage (e.g. a usb drive) once you finish using the PC.

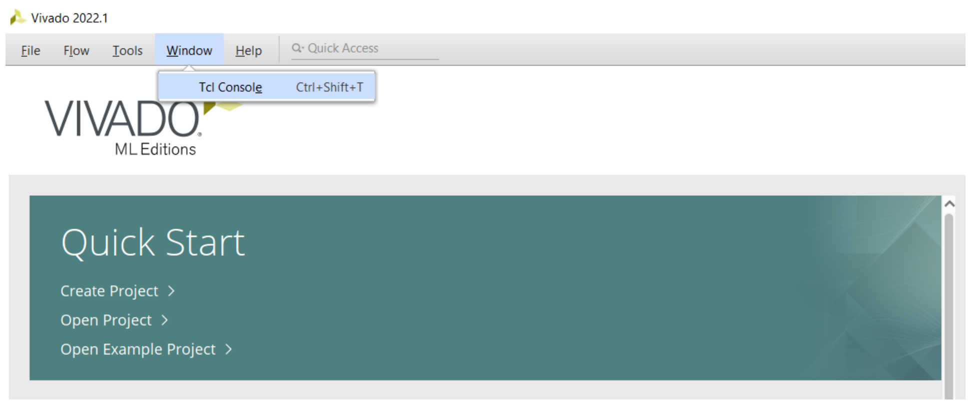


Figure 4.1: Tcl Console

- ☐ Launch Vivado 2022.1 by either:
  - opening Start Menu -> Xilinx -> Vivado 2022.1
  - going to `C:\Software\Xilinx\Vivado\2022.1\bin` and double clicking on the `vivado.bat` file.
- ☐ Open the Tcl console by clicking on `Window .. Tcl Console ..` as shown in Figure 4.1
- ☐ From the tcl console, navigate into the directory you placed your extracted files in by using the `cd <project_path>` command. Make sure to use forward slashes instead of back slashes in your paths because this is the only form of paths that Tcl accepts. For example, `cd C:/Users/d24lau/ECE423/ece423_prefab/`.
- ☐ To initialize the whole Vivado project run `source lab_prefab.tcl` in the Tcl console. This command creates a new Vivado project with the name `lab_prefab` in the same directory. It also generates a block design that includes all the needed hardware modules to run Assignment 1.

If it ever becomes necessary to restart the project from scratch, simply create another working directory and unzip the `ece423_prefab.zip` file into the new working directory and repeat the steps above.

4.2 Prefab Hardware Design

The block diagram of the generated prefab is shown in Figure 4.2. Notice that the blocks we are interested in for Assignment 1 are highlighted in red and are briefly described below:

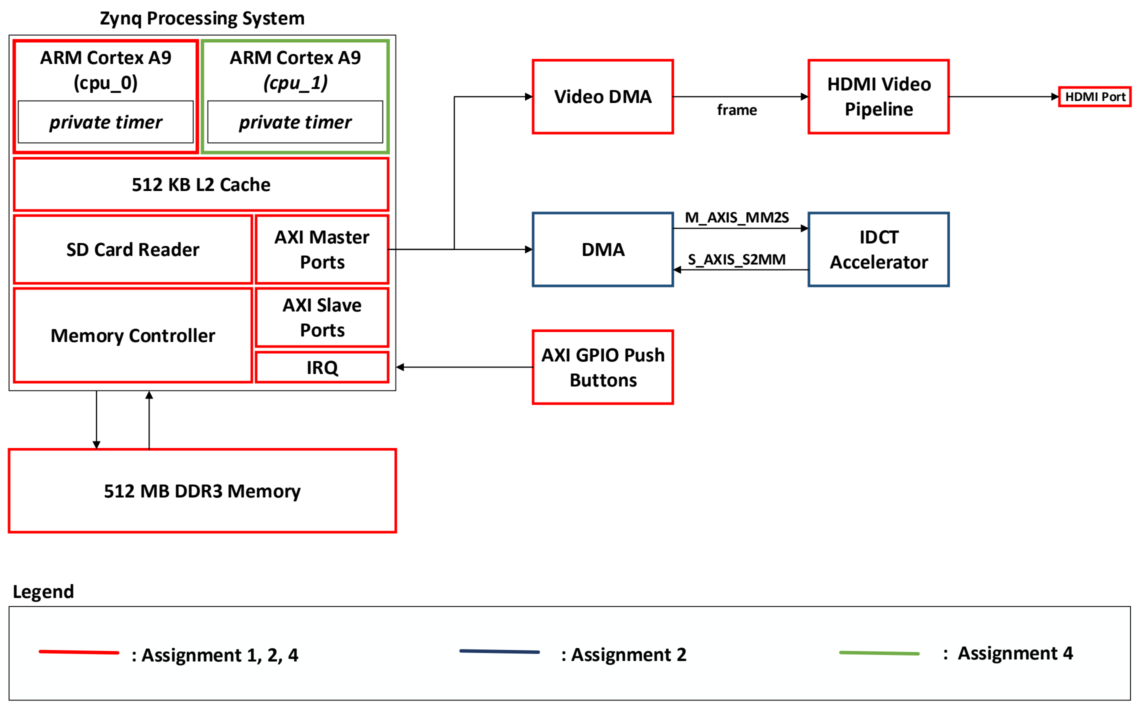


Figure 4.2: Prefab System Diagram

- The prefab contains a Zynq processing system which mainly constitutes the hardcore of the Pynq Z1 board. The detailed block diagram of the Zynq processing system can be seen in Figure 4.3. The Zynq blocks that we are mainly interested in are as follows:
  - Two **ARM cortex A9** processors `cpu_0`, and `cpu_1` running at 650 MHz. The first processor will be used to implement software application in Assignment 1, and the other one will be used for implementing the schedule in Assignment 4
  - **Private ARM timers** which will be used to set the desired frame rate
  - A **Memory controller** which will be used to store and fetch data from the available 512 MB DDR3 memory
  - An **SD Card Reader** peripheral which will be used to read the video files
  - A **512 KB L2 cache** which is shared among all processors

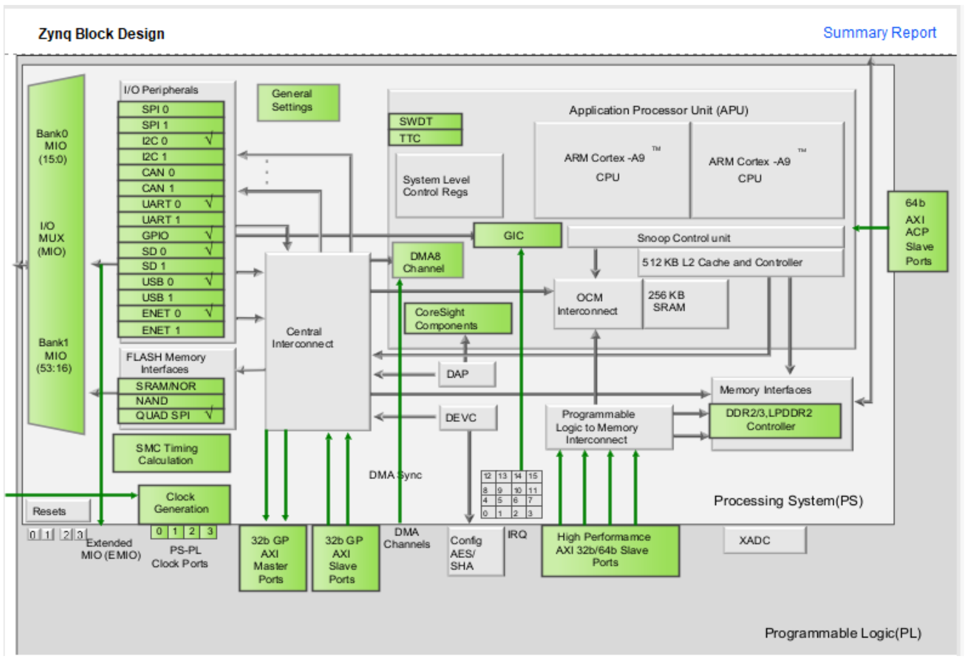


Figure 4.3: Zynq Block Design

- Two clock sources `FCLK_CLK_0` and `FCLK_CLK_1` running at **100 Mhz** and **142.5 Mhz** respectively.
- A **Direct Memory Access** module with read `(M_AXIS_MM2S)` and write `(S_AXIS_S2MM)` channels configured in loop-back mode. Both channels are connected to `FCLK_CLK_0` which is set to run at 100 MHz.
- A **Video Direct Memory Access (VDMA)** that has a master streaming interface `(M_AXIS_MM2S)` running at 142.5 MHz `(FCLK_CLK_1)`.
- An **HDMI Video Pipeline** component which converts the video frames streamed by the VDMA into video signals that can be displayed through the on-board HDMI port.
- An **AXI GPIO** component for the four push buttons that are found on the PYNQ board. This component invokes an interrupt whenever one of the buttons is pressed.
- Several **AXI Interconnect** interfaces that act as a switch and manage the traffic between different memory-mapped master and slave devices. This component allows any number of bus masters to communicate with any number of bus slaves, but only one master to slave communication request is processed at a time.

4.3 Video Signal Generation

Applications that require streaming video frames from main memory tend to be very bandwidth-intensive due to the volume of requests that need to be processed and the amounts of data that need to be transferred. Passing those memory requests through the main processor via software unnecessarily burdens the core and delays it from executing more urgent tasks, deteriorating the overall performance of the whole system. For that reason, a VDMA is used to process such memory requests without the involvement of the main processor. The VDMA generates a stream of frames that flows into the HDMI Video Pipeline and then gets displayed through the on-board HDMI port. The onboard HDMI port can display up to 60 frames per second which is equivalent to the refresh rate of the monitor that you will be using in the lab. However, in this project, you are required to run the application at 24 fps; hence, the frames will be displayed at a rate that is lower than the monitor's refresh rate.

The VDMA (`video_dma`) is clocked at 142.5 Mhz, while the video output is clocked at 74.25 Mhz which is the required pixel clock rate for 720p displays. For that reason, a dual-clock FIFO (`video_fifo`) is used to regulate the flow of pixels between the VDMA and the HDMI port. The VDMA is configured to perform unaligned transfers of 3 bytes per pixel, outputting a stream of 24-bit datasets to the HDMI Video Pipeline component. In more details, the video uses the RGB color format, where the 24 color bits (3 bytes) read by the VDMA for each pixel are arranged as follows: bits 7-0 are the blue component; bits 15-8 are the green component; bits 23-16 are the red component.

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

4.4 Generating the Bitstream (.bit) and the Hardware Definition (.xsa) files

To run the full software application on the Pynq board, you must first generate the bitsream file and the hardware definition file (.xsa file) of the project's block design. Follow the steps below to generate both files:

☐ After running the Tcl script and generating the block design of the whole project, open the block design in Vivado by expanding `lab_prefab_wrapper` in sources and double clicking on `lab_prefab_i` as shown in Figure 4.4.

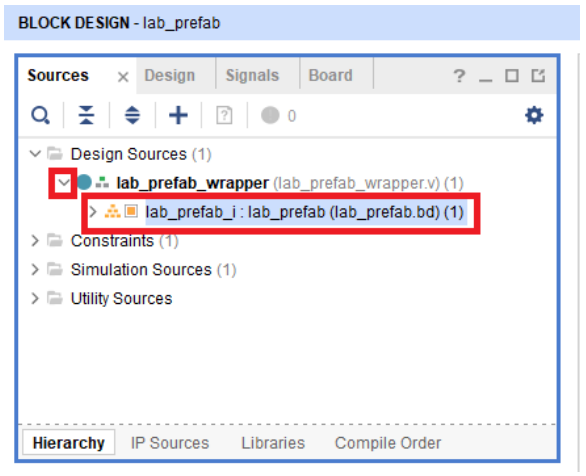
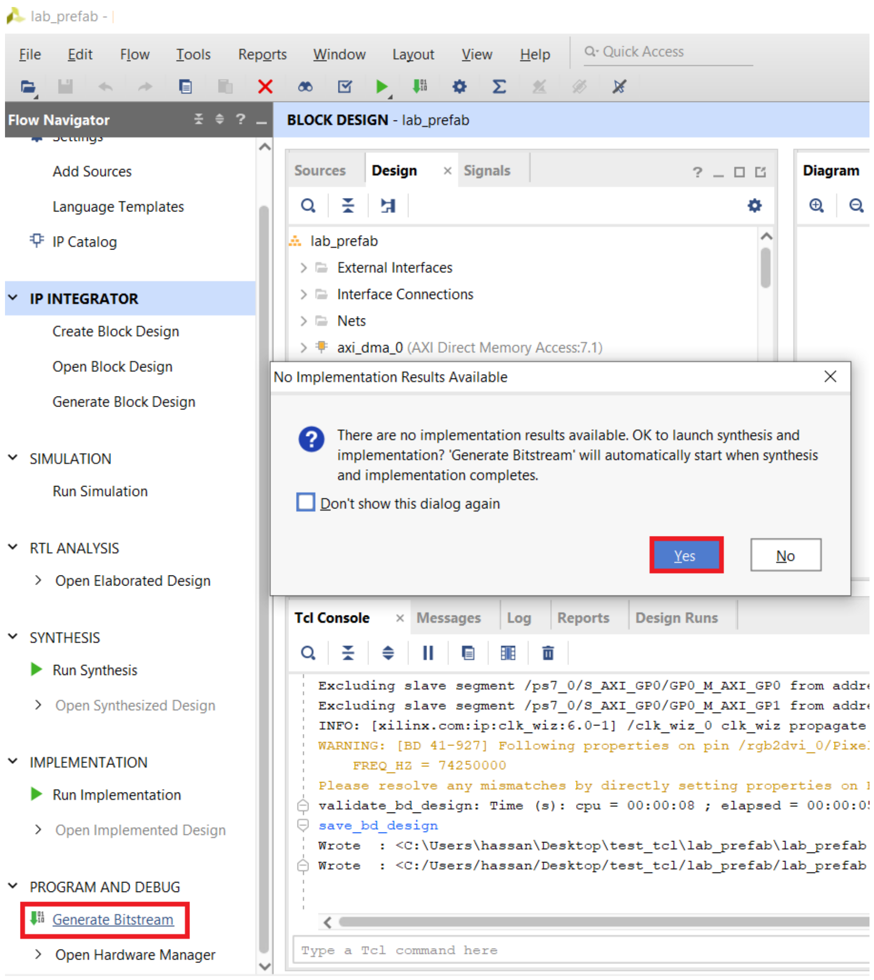


Figure 4.4: Opening the Block Design

- ☐ Verify that there are no errors in the design by clicking on `Tools ... Validate Design`.

- ☐ If the validation was successful and no errors were encountered, you can proceed to generate the bitstream of the design by clicking on `Generate Bitstram` in the `Program and Debug` section as can be seen in Figure 4.5. `Generate Bitstream` usually runs both synthesis and implementation and initially, none of those are there when we first generate the block diagram. That is why you should select Yes in case the dialogue box seen in Figure 4.5 pops up.



No Implementation Results Available

There are no implementation results available. OK to launch synthesis and implementation? 'Generate Bitstream' will automatically start when synthesis and implementation completes.

☐ Don't show this dialog again

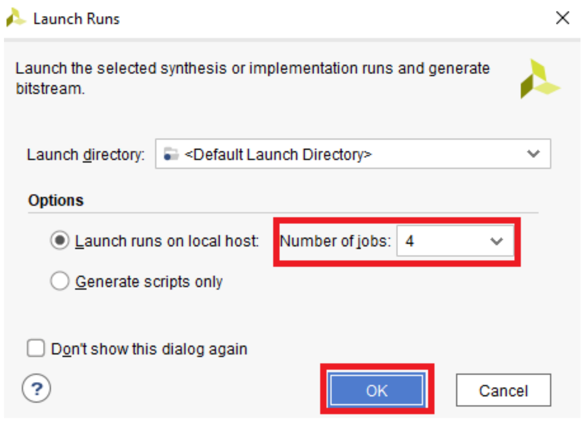
Yes No

```
Excluding slave segment /ps7_0/S_AXI_GP0/GP0_M_AXI_GP0 from addre
Excluding slave segment /ps7_0/S_AXI_GP0/GP0_M_AXI_GP1 from addre
INFO: [xilinx.com:ip:clk_wiz:6.0-1] /clk_wiz_0 clk_wiz propagate
WARNING: [BD 41-927] Following properties on pin /rgb2dvi_0/Pixe
    FREQ_HZ = 74250000
Please resolve any mismatches by directly setting properties on 
validate_bd_design: Time (s): cpu = 00:00:08 ; elapsed = 00:00:0
save_bd_design
Wrote  : <C:\Users\hassan\Desktop\test_tcl\lab_prefab\lab_prefab
Wrote  : <C:/Users/hassan/Desktop/test_tcl/lab_prefab/lab_prefab
```

Type a Tcl command here

Figure 4.5: Generating a bitstream

☐ In the next page select `Launch runs on local host` and set the number of jobs to 4 as shown in Figure 4.6. Note that generating the bitstream is a very compute-intensive task and it may take about 10 to 15 minutes to complete. You can monitor progress in the Design Runs tab in the console window.


Launch Runs

Launch the selected synthesis or implementation runs and generate bitstream.

Launch directory: <Default Launch Directory>

Options

Launch runs on local host: [✓] Number of jobs: 4

Generate scripts only

Don't show this dialog again

Figure 4.6: Selecting the number of jobs

☐ Upon completion, you will find the bitstream (.bit file) file in `project_directory\lab_prefab.runs\impl_1\lab_prefab.bit`

☐ Generate the Hardware Definition file (.xsa file) by clicking on `File ... Export ... Export Hardware` as can be seen in Figure 4.7

☐ Select the `Include Bitstream` option and click Next.

☐ Leave the default XSA file name as `lab_prefab_wrapper` and the default export folder as `project_directory/lab_prefab`. Click `Finish`.

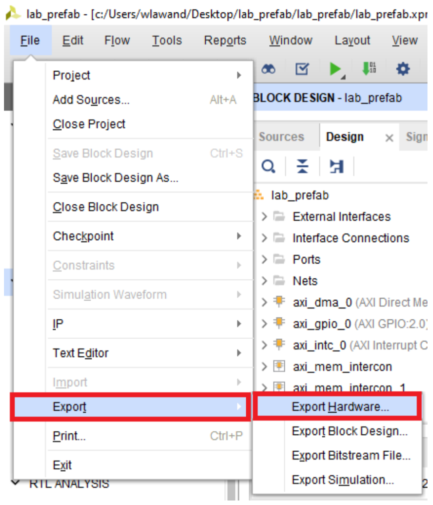


Figure 4.7: Generating the .xsa file

Note that in Figure 4.2, we have 2 ARM processors (`cpu_0 and cpu_1`) within the Zynq processing system. However, for lab 1, we will only be using `cpu_0`.

4.5 Creating a Software Application

Once you have generated the bitstream and hardware definition file, you can create a software application to run on the zynq processing system. Follow the steps below to create a single-core application running on `cpu_0`.

4.5.1 Setting up Vivado Project

1. Make sure the jumper JP4 is correctly placed to JTAG position as marked on the board.

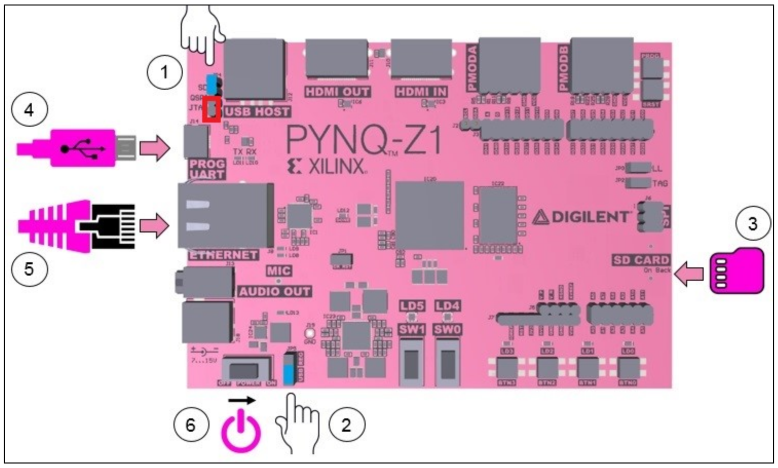
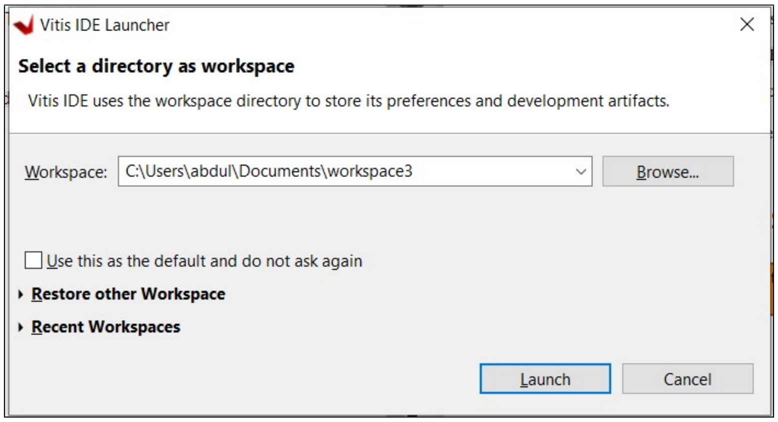


Figure 4.8: PYNQ-Z1 programming mode (JP4) jumper

2. Start Vitis 2022.1 and set a workspace folder that is different than the Vivado project folder. A recommended location is `project_directory\workspace`. Click the "Launch" button.


Vitis IDE Launcher

Select a directory as workspace

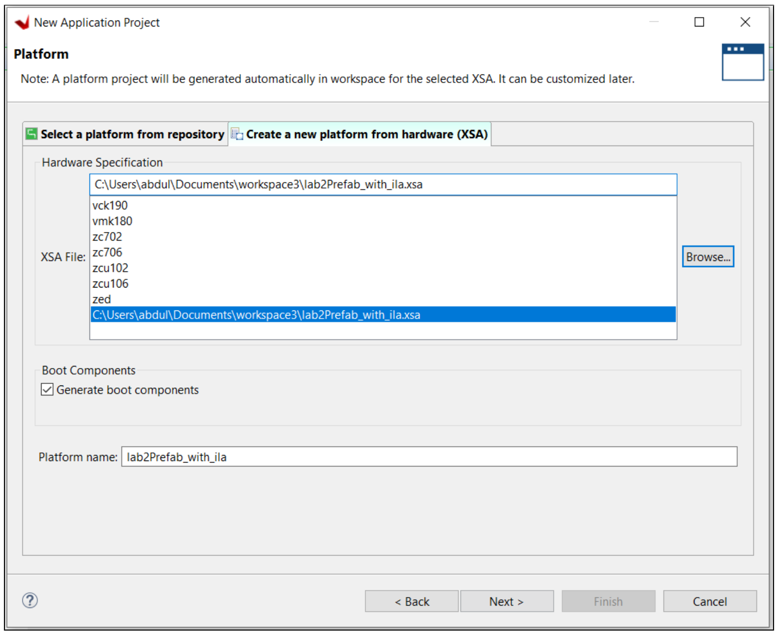
Vitis IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\Users\abdul\Documents\workspace3

Use this as the default and do not ask again

Figure 4.9: Workspace

3. Select "Create Application Project". If shown a Welcome Page, click 'Next'.
4. On the screen for "Platform", select the tab for "Create new platform from hardware (XSA)".
5. Click the 'Browse ...' button. Select the XSA file that was exported from Vivado. This should be `project_directory/lab_prefab/lab_prefab_wrapper.xsa`. Click the "Next" button.


New Application Project

Platform

Note: A platform project will be generated automatically in workspace for the selected XSA. It can be customized later.

Select a platform from repository | Create a new platform from hardware (XSA)

Hardware Specification

XSA File: C:\Users\abdul\Documents\workspace3\lab2Prefab_with_ila.xsa

```
vck190
vmk180
zc702
zc706
zcu102
zcu106
zed
C:\Users\abdul\Documents\workspace3\lab2Prefab_with_ila.xsa
```

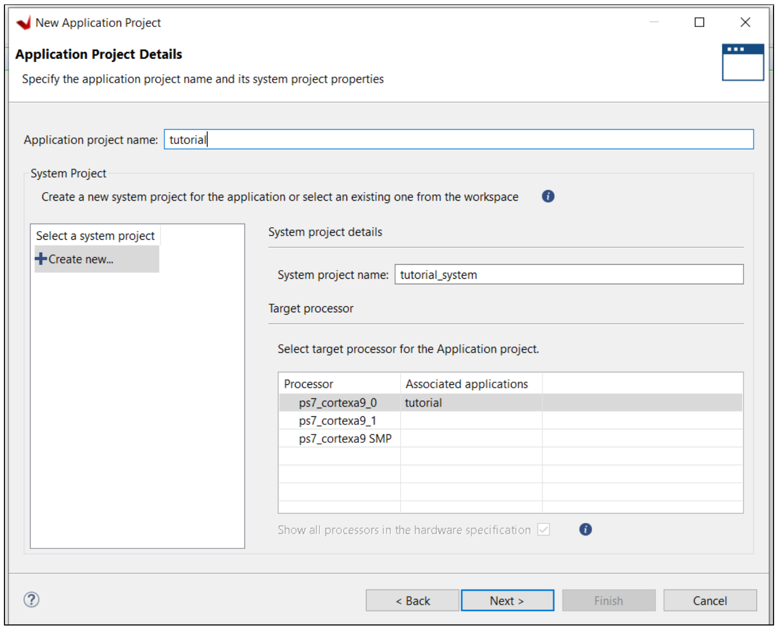
Boot Components

☑ Generate boot components

Platform name: lab2Prefab_with_ila

Figure 4.10: Hardware platform

6. Enter an application project name; for example `hello_world`. Note that the application project name automatically is associated with the "ps7_cortexa9_0" processor. The ZYNQ processor has two cores, cpu0 and cpu1. This assigns the code to run on the cpu0 core. Click "Next".



Application Project Details

Specify the application project name and its system project properties

Application project name: tutorial

System Project

Create a new system project for the application or select an existing one from the workspace

Select a system project

Create new...

System project details

System project name: tutorial_system

Target processor

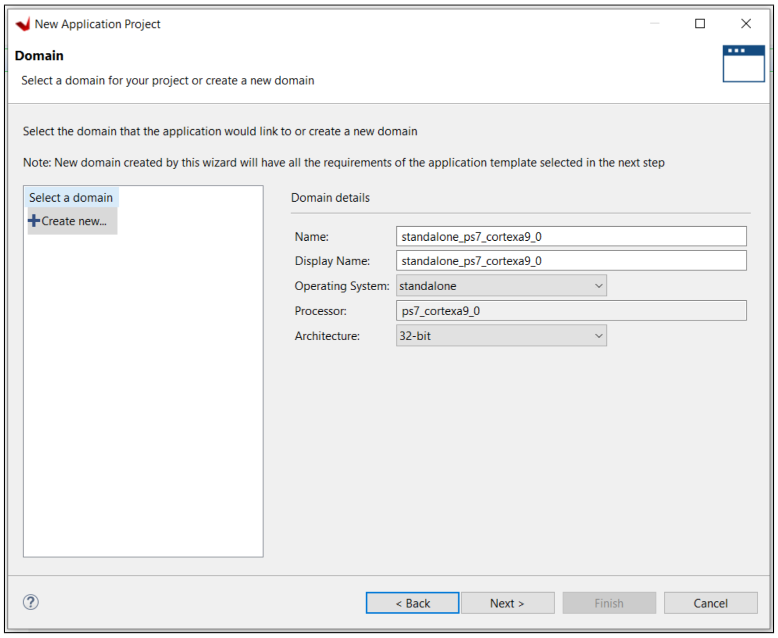
Select target processor for the Application project.

Processor	Associated applications
ps7_cortexa9_0	tutorial
ps7_cortexa9_1	
ps7_cortexa9 SMP	

Show all processors in the hardware specification ✓

Figure 4.11: Application Project Name

7. On the domain page, you do not need to change anything. For now, you can keep the options as shown in figure below. Click Next.



New Application Project

Domain

Select a domain for your project or create a new domain

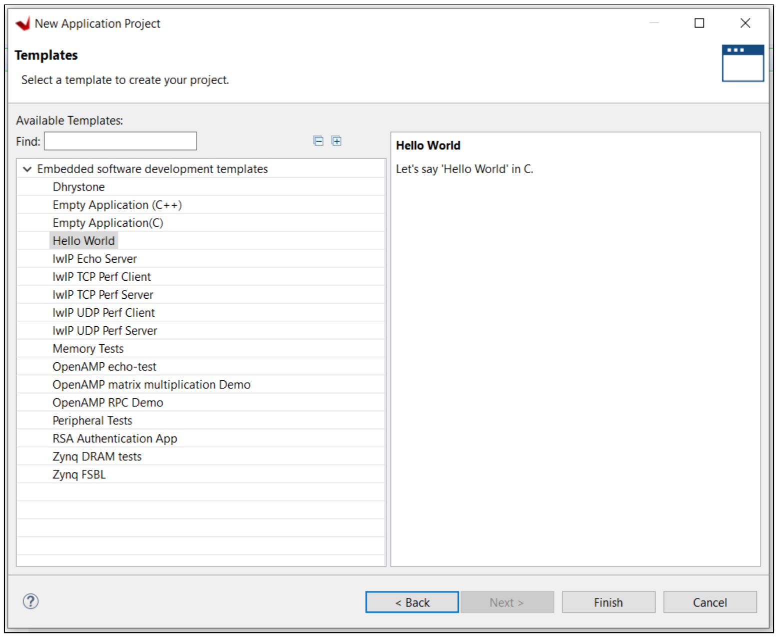
Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Name: standalone_ps7_cortexa9_0
Display Name: standalone_ps7_cortexa9_0
Operating System: standalone
Processor: ps7_cortexa9_0
Architecture: 32-bit

Figure 4.12: Domain

8. Select the "Hello World" template, as shown in figure below. Click "Finish".



New Application Project

Templates

Select a template to create your project.

Available Templates:
Find:

Embedded software development templates
- Dhrystone
- Empty Application (C++)
- Empty Application(C)
- Hello World
- lwIP Echo Server
- lwIP TCP Perf Client
- lwIP TCP Perf Server
- lwIP UDP Perf Client
- lwIP UDP Perf Server
- Memory Tests
- OpenAMP echo-test
- OpenAMP matrix multiplication Demo
- OpenAMP RPC Demo
- Peripheral Tests
- RSA Authentication App
- Zynq DRAM tests
- Zynq FSBL

Hello World

Let's say 'Hello World' in C.

Figure 4.13: Selecting Hello World template

9. The hello_world application should now be created. To explore, feel free to expand the "src" folder and open the "helloworld.c" file.

4.5.2 Creating Run Configuration

1. **Creating Run Configuration:** In the toolbar, click on the dropdown next to "Run" button to open the Run Configuration tab.

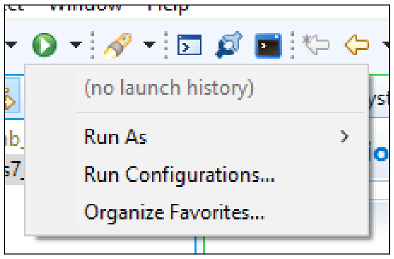
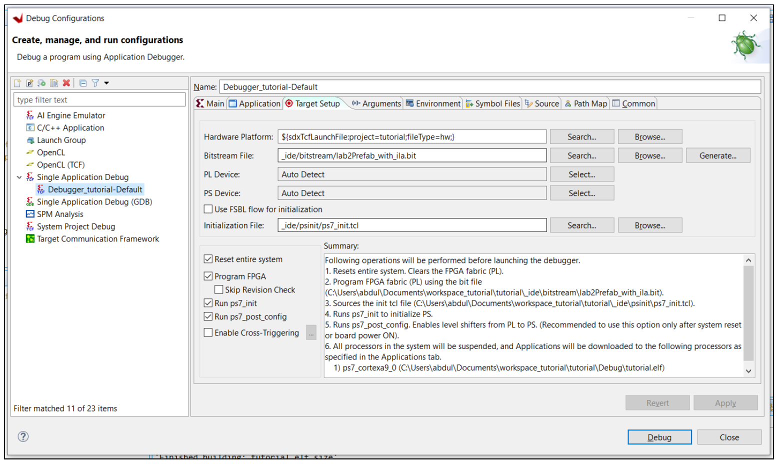


Figure 4.14: Run button

2. Double click on single application debug to create a configuration. In the Main tab, change the Configuration from Debug to Release (Note, in release mode, the program is compiled with compiler optimizations which makes it difficult to use the debugger tool, but it gives more accurate timing results, switch back to Debug mode, when using Vitis Debugger Tool). In the Target Setup tab, select the path to the appropriate bitfile. Note that the standard bitfile for the lab includes configuration for UART, that is used to run the print statements in helloworld application. Leave the other settings as default.



Debug Configurations

Create, manage, and run configurations

Debug a program using Application Debugger.

- AI Engine Emulator
- C/C++ Application
- Launch Group
- OpenCL
- OpenCL (TCF)
- Single Application Debug
  - Debugger_tutorial-Default
- Single Application Debug (GDB)
- SPM Analysis
- System Project Debug
- Target Communication Framework

Filter matched 11 of 23 items

Name: Debugger_tutorial-Default

Main | Application | Target Setup | Arguments | Environment | Symbol Files | Source | Path Map | Common

Hardware Platform: ${sdxTcfLaunchFile:project=tutorial;fileType=hw;}

Bitstream File: _ide/bitstream/lab2Prefab_with_ila.bit

PL Device: Auto Detect

PS Device: Auto Detect

☐ Use FSBL flow for initialization

Initialization File: _ide/psinit/ps7_init.tcl

☑ Reset entire system
☑ Program FPGA
☐ Skip Revision Check
☑ Run ps7_init
☑ Run ps7_post_config
☐ Enable Cross-Triggering

Summary:

Following operations will be performed before launching the debugger.
1. Resets entire system. Clears the FPGA fabric (PL).
2. Program FPGA fabric (PL) using the bit file (C:\Users\abdul\Documents\workspace_tutorial\tutorial_ide\bitstream\lab2Prefab_with_ila.bit).
3. Sources the init tcl file (C:\Users\abdul\Documents\workspace_tutorial\tutorial_ide\psinit\ps7_init.tcl).
4. Runs ps7_init to initialize PS.
5. Runs ps7_post_config. Enables level shifters from PL to PS. (Recommended to use this option only after system reset or board power ON).
6. All processors in the system will be suspended, and Applications will be downloaded to the following processors as specified in the Applications tab.
   1) ps7_cortexa9_0 (C:\Users\abdul\Documents\workspace_tutorial\tutorial\Debug\tutorial.elf)

Figure 4.15: Setting up run/debug configuration

3. **Optional: optimization flags for Release mode:** Note that release configuration compiles the project with -O2 optimizations by default. The optimization level can be controlled by right clicking on the project name in the project explorer on the left and going to Properties ->C/C++ Build ->Settings (making sure the configuration is set to Release) and then in the Tool Settings tab clicking on ARM v7 gcc compiler ->Optimization. From here, you can experiment by setting the optimization level to -O3 and/or adding specific optimization flags. You might have learned from previous courses that the effects of gcc optimization flags on the execution time of a program is hard to predict - some functions run faster when switching to -O3, and some run slower. Depending on your code, your schedule in Assignment 3, etc., you might find that -O3 is either beneficial or detrimental. If you do not have experience with this type of code optimization, it is ok to leave the flag to -O2.

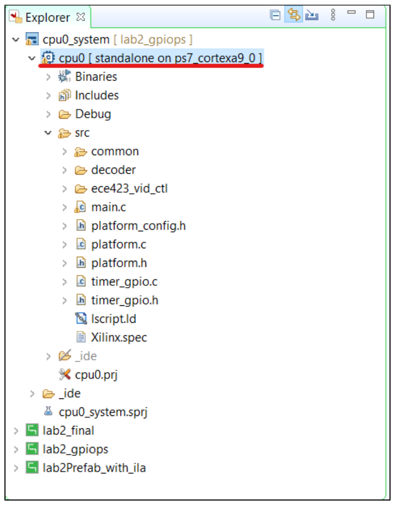
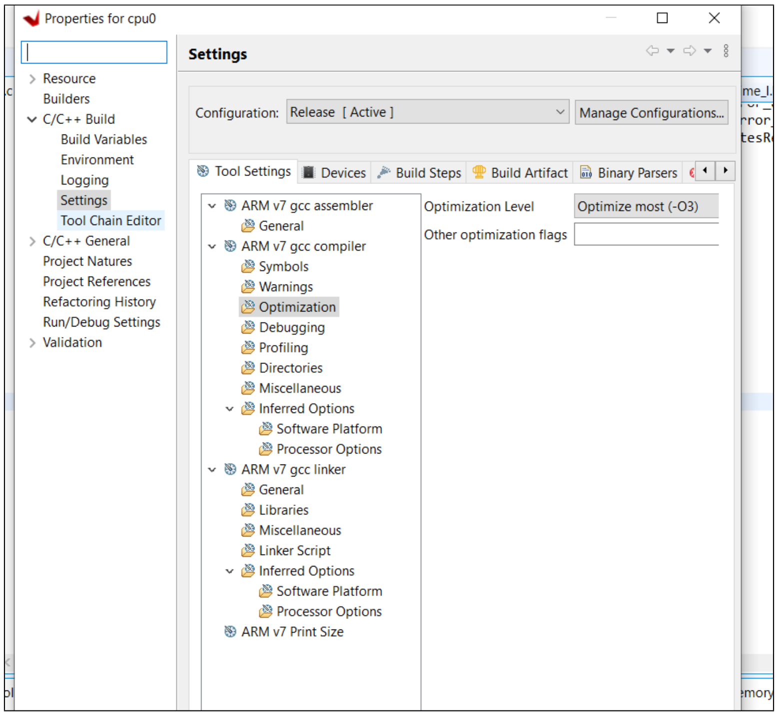


Figure 4.16: Configuring Optimization Flag



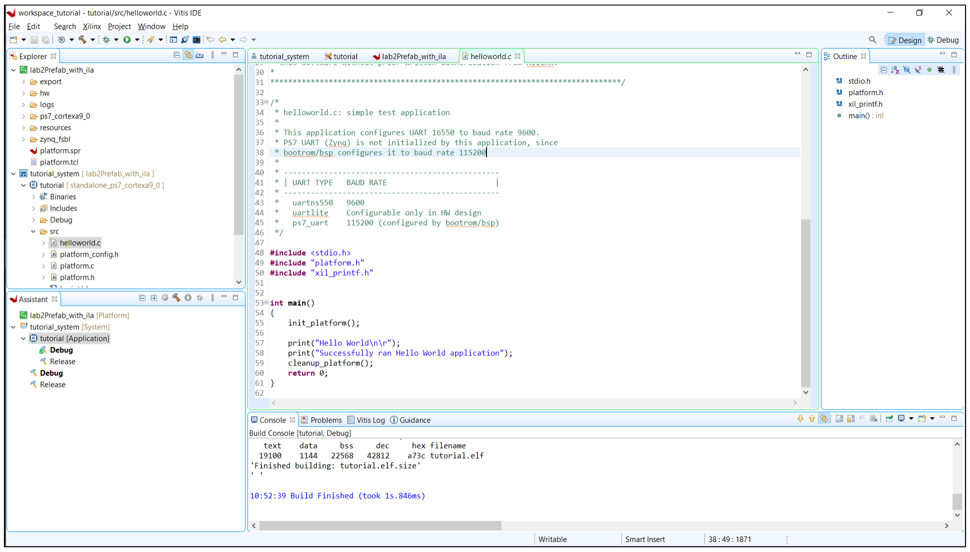
Configuration: Release [Active]

Optimization Level: Optimize most (-O3)

Other optimization flags:

Figure 4.17: Configuring Optimization Flag

4. **Configuring build options:** Open the helloworld.c file to view the source. To configure build modes, use the dropdown next to Build button in the toolbar at the top. In the dropdown next to the build button, you can choose to build in Debug or Release mode. For timing analysis, it is required to build in Release mode.



```
*
*****************************************************************************/

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}
```

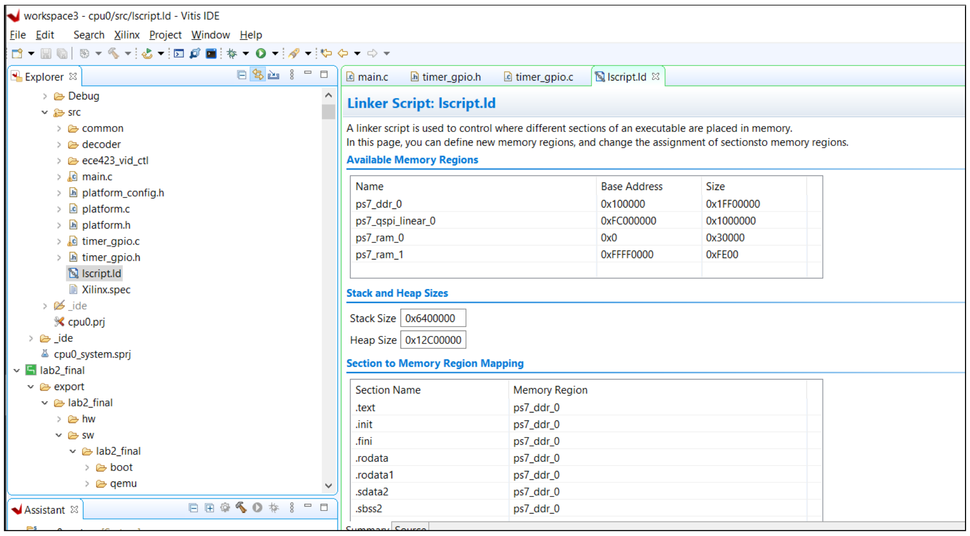
Build Console [tutorial, Debug]

```
   text    data     bss     dec     hex filename
  19100    1144   22568   42812    a73c tutorial.elf
'Finished building: tutorial.elf.size'
' '

10:52:39 Build Finished (took 1s.846ms)
```

Figure 4.18: Building the project

5. **Configuring linker script:** You also need to adjust the stack and heap sizes in linker script (lscript.ld). In particular, the default size of the heap will not be sufficient to allocate all required data structure by the MJPEG423 decoder application.



Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory.
In this page, you can define new memory regions, and change the assignment of sectionsto memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0	0x100000	0x1FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFF0000	0xFE00

Stack and Heap Sizes

Stack Size 0x6400000

Heap Size 0x12C00000

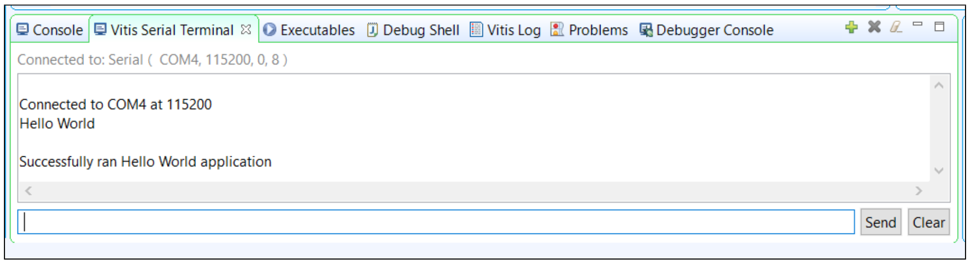
Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.sdata2	ps7_ddr_0
.sbss2	ps7_ddr_0

Figure 4.19: Building the project

6. **Running the application:** you can run the application in Debug or Release configuration. Make sure the correct configuration is selected and the build for that configuration is updated.

7. You can view the output on the serial terminal of your choice. Vitis provides a serial terminal which is available in Window → Debug Perspective. It provides a list of COM Ports. The default baud rate for the helloworld application is 115200. Note that the Vitis terminal must be reconfigured every time you reset the board - for this reason, you might find using Putty to be more convenient instead.



```
Connected to: Serial ( COM4, 115200, 0, 8 )

Connected to COM4 at 115200
Hello World

Successfully ran Hello World application
```

Figure 4.20: Viewing the output

4.6 Task 1: Execute the sequential MJPEG423 decoder

Your goal in this task is to execute the MJPEG423 decoder on a **single ARM Cortex A9 core** (i.e., sequentially). 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 3. Use the following hints to make the necessary changes:

1. The provided MJPEG423 sample application contains both the encoder and decoder. **You should not try to run the encoder on the Cortex A9** - instead, make sure you only use the code of the decoder. Start by creating a single-core software application following the instructions in Section 4.5. Then copy the `decoder` and `common` folders from the provided `mjpeg423app` (in Learn) to the `src` folder of your application - you can do this by dragging and dropping the folders from file explorer into Vitis. We suggest you to modify the `mjpeg423_decoder.c` file to implement the required functionality, calling the `mjpeg423_decoder` function from main.

2. The sample decoder reads a video file using file handling functions in C. Instead, we need to read the video files from the SD Card. To do so, you can use the FatFs library, which provides functionalities similar to the file handling functions in C; refer to Appendix A.1 for information on how to include and use the library. The video files are in the root directory of the third partition of the SD card. In particular, the SD card contains video files of progressively larger duration and hence file size: `v1_300.mpg` is the smallest video with 300 frames and `v1_1730.mpg` is the largest video with 1730 frames. You can begin

testing the application using the first (smallest) file, but by the end of Lab 1 you should be able to decode all provided files.

3. The sample decoder 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). Instead, we want to direct the output to the hardware video module through the VDMA. To simplify your job of controlling the VDMA, we provide a video library; you can find it on Learn in the libraries.zip file. Start by copying the content of the zip file to your src directory. A description of the video API is provided in Appendix A.2. As discussed in Section 4.3, the VDMA takes the information for each frame from a specified location in main memory, called the frame buffer. To output a new video frame, you should first obtain a frame buffer pointer using `buff_next`; then write the frame content to the frame buffer, register it using `buff_reg`, and finally inform the DMA to display the next registered frame using `vdma_out`. The actual number of buffers can be specified when initializing the video system using `vdma_init`. As discussed in Section 4.3, you must set frame_buff_size = 2 to use double-buffering: one buffer is used by the VDMA, while the other buffer is used to process the next frame. When setting frame_buff_size ≥ 3, it is actually possible to register multiple buffers before outputting the next frame; this is not useful in Assignment 1, but it will be needed in Assignment 4.

4. You are only required to decode videos of size 1280x720 pixels, since this is the size of the frame buffer used by the video controller in the hardware.

5. The `ycbcr_to_rgb` function in the sample application converts the colors from Y'CbCr format to RGBA format used by bmp images (see Section 3.2), where each pixel is represented on 32 bits (see the `rgb_pixel_t` structure in `mjpeg423_types.h`). However, for our project, the HDMI component accepts the video in RGB format as discussed in Section 4.3, which uses 24 bits per pixel; hence, you need to modify the function to remove the alpha component. This also changes the size of the framebuffer.

6. Since a single ARM A9 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 Assignments 1 and 2.

4.7 Task 2: Implement Playback Control with Periodic Timer

Improve your application by implementing playback control and periodically displaying the frames. The specification of the required playback control system is provided below, where `BTN0`-`BTN3` refer to four pushbuttons on the pynq board:

1. In the `paused` state, the application should display a single frame of a video. In the `playing` state, the application should play a video (display each successive frame of the video at a **constant frame rate**).

2. Upon starting, the application should load the first valid video in the root directory of the SD card, display its first frame and transition to the `paused` state. A valid video has extension `.mpg`. Pressing `BTN0` while in either `paused` or `playing` state should cause the application to load the next valid video, display its first frame and transition to the `paused` state. If the application has reached the last valid file in the root directory, pressing `BTN0` again should cause it to cycle back to the first valid file.

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

4. After playing the last frame in a video, the application should transition to `paused`.

5. Pressing `BTN1` switches between the `paused` and `playing` state; unless the application has reached the last frame in a video, in which case it does nothing.

6. Pressing `BTN2`/`BTN3` while in the `paused` or `playing` state should cause the video to skip forward/backward by F frames, where F is **roughly** 120 frames (5 seconds), and remain in the same state. If you skip forward and the remaining number of frames in the video is less than F, then the application should simply transition to `paused` on the current frame (or do nothing, if it is already `paused`). If you skip backward when you are less than F frames into the video, you should restart from the first frame of the video.

Notes

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

2. The sample MJPEG423 decoder reads the trailer of the file upon starting. As discussed in Chapter 3, the trailer contains the list of I-frames (frame number and position in the file) for the video file. You can implement `BTN2`/`BTN3` functionality by keeping track of the index of the current frame; when the user wants to skip behind/ahead, simply look for an I-frame that is roughly 120 frames ahead/behind the current frame and restart from that frame.

3. To display frames periodically, you need a timer. To simplify implementation, we provide a library that supports interrupt handling for the ARM SCU timer and the AXI GPIO

to which pushbuttons are connected; the library is provided in file libraries.zip on Learn and its specification is provided in Appendix A.3. Note that `vdma_out` should be called from the TimerISR. The function returns 0 if there is no registered frame to display. To allow the lab team to check that the video is displayed at the correct frame rate (which is important in Assignment 4), you must be able to detect this error in your code and print a suitable message.

4. Since the application runs very slowly in Assignment 1, you will have to set the timer period to a value much higher than 41.66 ms (i.e., run the application much slower than 24 fps). This is fine for Assignment 1 and 2. Specifically, you can simply set the timer period based on the slowest (most computationally intensive) frame in any video, so that you are guaranteed to be able to process any one frame in one period.

4.8 Task 3: Profile the Application

In order to optimize the system design, you need to collect essential performance metrics. In particular, how the various components of the system behave. Since our system is (relatively) simple, in terms of behavior we will only be interested in execution time and main memory (DDR3) utilization.

4.8.1 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 Assignment 1 you will write profiling code to measure the time required to execute certain functions (the functional blocks of the MJPEG423 decoder).

To measure time, we use the ARM SCU Timer. To get the value of absolute time, xilinx provides interface to 64-bit Global Counter in the PMU. This counter increases by one at every two processor cycles. The `xtime_l.h` file provides the necessary functions to perform the time profiling. `XTime_GetTime` function is used to get absolute time at any point in the program. This function takes pointer to a variable of type `XTime` which is 64 bit unsigned decimal type, to store the absolute time value. This function can be called multiple times to compute the time difference.

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

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

4.8.2 Memory Utilization

In Assignment 3, it will become important to ensure that the data allocated by each core fits in the available main memory. Memory usage is dominated by the data structures that hold the data used as input and output by each function block. Hence, you are required to analyze the size of such structures as part of Assignment 1. 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 writes an 8x8 pixel block as output. The `lossless_decode` is more complex since it depends on the size of the bitstreams, which is variable based on the frame; as explained earlier, you thus need to determine at least average, minimum and maximum size. Note that bitstream size information is provided for each frame in a MJPEG423 file.

4.8.3 Task Requirements

Profile the decoder application based on the former discussion. In particular, you must profile the following functional blocks from `v1_1730.mpg`: 1. the code used to read the file from SD, one frame at a time; 2. `lossless_decode`; 3. `idct`; 4. `ycbcr_to_rgb`. 5. `buff_reg`. Other video / timer / GPIO functions do not need to be profiled since they take negligible time. For each block, report timing and memory utilization information.

For SD read and `lossless_decode`, you must distinguish between P and I-frames, since the two have quite different times. For `lossless_decode`, you also need to profile the time and memory utilization for each color component (Y / Cb / Cr). You must report maximum, average and minimum times. For `idct` and `ycbcr_to_rgb`, there is no significant difference between frames and color components, so simply compute the time and memory utilization required to process all blocks in a frame.

Note you will need to perform a similar profiling in Assignment 2 after changing parts of the application, so make sure your profiling code can be re-used later on. For example, you can enclose the profiling code within conditional compilation directives - note that the sample MJPEG423 application already uses conditional compilation to enable/disable debug information.

4.9 Deliverables

Demo: during the demo, you must first demonstrate the decoding application running on one of the cortex A9 cores without the periodic timer (i.e., decoding each frame one after the other), and then using the timer in conjunction with playback control. You must be able to decode all files provided on the SD card, but no profiling is required at this point. Submit to Learn all decoder files (in an archive) that you modified as part of the assignment.

Report: your report must include tables reporting all required application profiling and memory utilization information. A template describing the required format for the report is provided on Learn.

The mark breakdown is reported in Table 4.1.

Total Assignment 1	8%
Demo	5%
Report	3%

Table 4.1: Mark Breakdown For Assignment 1

Chapter 5

Assignment 2

In Assignment 2, you will create a hardware accelerator to speed-up the execution of the most critical part of the application - the inverse discrete cosine transform (IDCT). You will also profile the performance of the hardware components.

5.1 System-level Design

A simplified block diagram of the employed hardware architecture is presented in Figure 4.2. For this assignment you will use the DMA IP that is provided by Xilinx and that has read and write channels. Initially, this block is configured to work in a loop-back mode which means that the write channel is connected to the read channel. With this configuration the DMA is used to perform a memory-to-memory copy. For this assignment, we need to use the read channel of the DMA to fetch the 8x8 blocks of dequantized coefficients (`YDCAC`, `CbDCAC` or `CrDCAC` matrix) from main memory and pass them to the IDCT Accelerator through the `M_AXIS_MM2S` streaming interface. The streaming interface is a simple synchronous interface designed for point-to-point connection of hardware IP blocks. The IDCT Accelerator computes the 2D IDCT on each incoming 8x8 block of dequantized coefficients, and outputs a corresponding 8x8 pixel color block (either for Y', CB or CR color component) to the main memory through the `S_AXIS_S2MM` streaming interface.

As part of Assignment 2, you will implement the IDCT Intellectual Property module (IP) and connect its input and output to the `M_AXIS_MM2S` and `S_AXIS_S2MM` streaming interfaces respectively. In what follows, Section 5.1.1 discusses how to configure and program the DMA core, while Section 5.1.2 discusses required cache management operations.

5.1.1 DMA Core

As discussed at the beginning of the chapter, you will need a single DMA core with a read and a write channel (other than the video DMA) for the accelerator design. Both channels are configured with a memory-mapped interface to read and write data to the DDR3 memory through the `M_AXIS_MM2S` and `S_AXIS_S2MM` streaming interfaces. The read channel has a 32-bit streaming interface and should be configured in software to read multiple full IDCT blocks, each of which has size 1024 bits (8*8 16-bit elements). The write channel has the same streaming width but should be configured to write the same number of full-color block, each of which has size 512 bits (64 bytes). **Note** that the DMAs are configured with `Full Word Accesses Only` to simplify the hardware used inside the DMA by avoiding unaligned words. In software, simply using malloc() to allocated RGB blocks would ensure the array is aligned correctly in memory (i.e. 8 byte alignment).

The DMA driver API is detailed at `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842100/AXI+DMA+Standalone+Driver`. We suggest to program the DMA in SimpleTransfer mode using polling. In this mode, the user sends a single DMA descriptor at a time to each DMA channel. Following is the list of functions (in-order) you need to use to operate the DMA in this mode. Sample usage of these functions is given in example provided with the driver `https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/axidma/examples/xaxidma_example_simple_poll.c`:

- `XAxiDma_LookupConfig`: this function looks up the DMA config from provided hardware description and provides a pointer to XAxiDma_Config structure.
- `XAxiDma_CfgInitialize`: this function initializes the DMA with the provided DMA config.
- `XAxiDma_IntrDisable`: this function disables the interrupt for the provided channel of DMA. This function should be called for both read and write channels of DMA.
- `XAxiDma_SimpleTransfer`: this function initiates a transfer of either channel of DMA. Use this function to send and receive the data through DMA.
- `XAxiDma_ReadReg`: this function can be used to read the DMAIntErr bit of Rx channel to see if the transer is completed.
- `XAxiDma_Reset`: this function is used to reset the DMA. This should be performed after every pair of read and write requests.
- `XAxiDma_ResetIsDone`: this function should be used to check if DMA core has successfully exited out of reset.

Below is a sample listing of a program using DMA API. Note that you will have to modify the code to specify the desired size and address of the read and write data structures in main memory in the `XAxiDma_SimpleTransfer` calls.

```
#include <xaxidma.h>
XAxiDma *InstancePtr;
XAxiDma AxiDma;

void main(){
    // Initialize the DMA
    InstancePtr = &AxiDma;
    XAxiDma_CfgInitialize(&AxiDma, XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID));
    XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DMA_TO_DEVICE);

    //Perform a transfer
    XAxiDma_SimpleTransfer(&AxiDma, , , XAXIDMA_DEVICE_TO_DMA); //Program write channel
    XAxiDma_SimpleTransfer(&AxiDma, , , XAXIDMA_DMA_TO_DEVICE); //Program read channel

    //Poll until transfer is done, then reset the DMA in preparation for next transfer
    while (!(XAxiDma_ReadReg(InstancePtr->RxBdRing[0].ChanBase, XAXIDMA_SR_OFFSET) &
        XAXIDMA_ERR_INTERNAL_MASK)) {}
    XAxiDma_Reset(&AxiDma);
    while(!XAxiDma_ResetIsDone(&AxiDma)){}
}
```

5.1.2 Cache Management

The ARM Cortex A9 cores in the Zynq processing systems support private L1 write-back data and instruction caches and a single shared L2 cache. A hardware cache coherence mechanism ensures that the content of the L1 caches is kept synchronized between the cores - if a core modifies the content of a cache block in its private L1, such modification will be seen by the other core.

However, coherency is not maintained between the A9 caches and I/O devices, including the DMA core. This creates two potential problems:

1. Since the data cache is write-back, whenever a core writes to a 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 L2 cache. Therefore, if the DMA tries to read the buffer after the core 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.

2. After the DMA writes to a buffer in main memory, the core 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 cache before data was modified (for example, while processing the previous frame).

The Xilinx SKD provides a cache management API that can be used to address both issues (see Appendix A.4). (1) To solve the first issue, after writing to the buffer, the core must *flush* the buffer from cache. A flush request forces the cache to write-back all modified cache blocks and then invalidate them. This ensures that the modified buffer content is written to main memory. (2) To solve the second issue, before reading from the buffer, the core must invalidate it from cache. This ensures that the modified buffer content is read from main memory.

Note 1: the API provides functions to either flush / invalidate the whole cache, or only a specific range of addresses. The latter work by stepping through each cache block in the specified range, and requesting the cache to flush / invalidate it. For this reason, if you need to flush / invalidate a large buffer (hundreds of more cache blocks), it is recommended to simply flush / invalidate the whole cache. Also keep in mind that flush means write-back + invalidate. For this reason, if you flush the whole cache before performing a DMA transfer, you do not need to also invalidate it after the transfer completes, since the flush operation already invalidates cache blocks of the DMA write buffer before the core starts reading it. Finally, note that since there are two cache levels, you need to first flush / invalidate L1 and then L2.

Note 2: the VDMA used by the video component is similarly not coherent with the Cortex A9 caches. Hence, before a framebuffer can be displayed by the VDMA, it must be flushed from cache. In the provided video display library, this is performed by flushing the whole L1 and L2 inside the `buff_reg` function - this is the reason `buff_reg` takes a non-neglibile amount of time and we asked you to profile it.

5.2 Task 1: Create, Test and Optimize the HLS Accelerator

As part of this task, you will create and test the IDCT hardware accelerator using a High-Level Synthesis (HLS) flow under Vitis HLS.

5.2.1 Getting Started with HLS

The general design flow of the HLS tool is shown in Figure 5.1. In this assignment, you will go through most of these steps. Each step is further elaborated on in the coming sections.

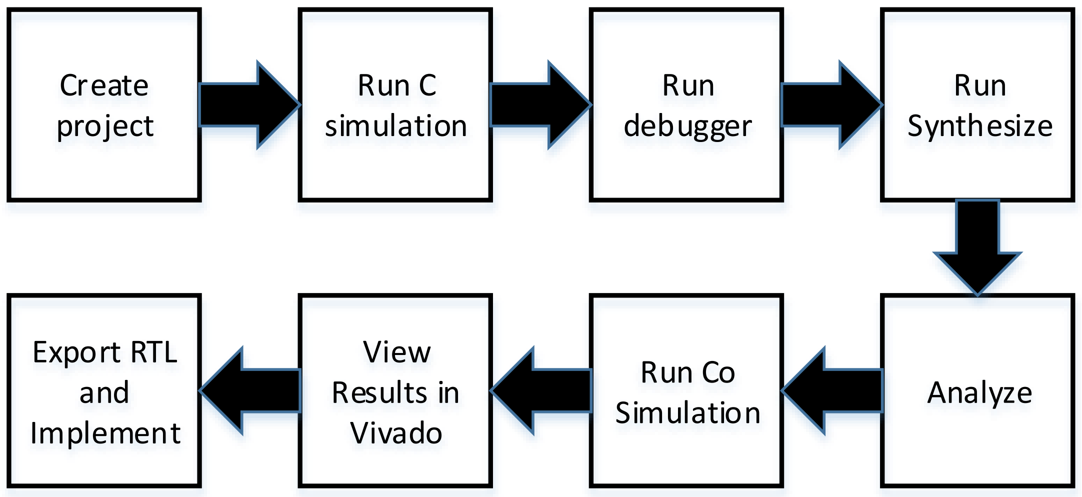


Figure 5.1: General Flow of HLS.

5.2.2 Create Project

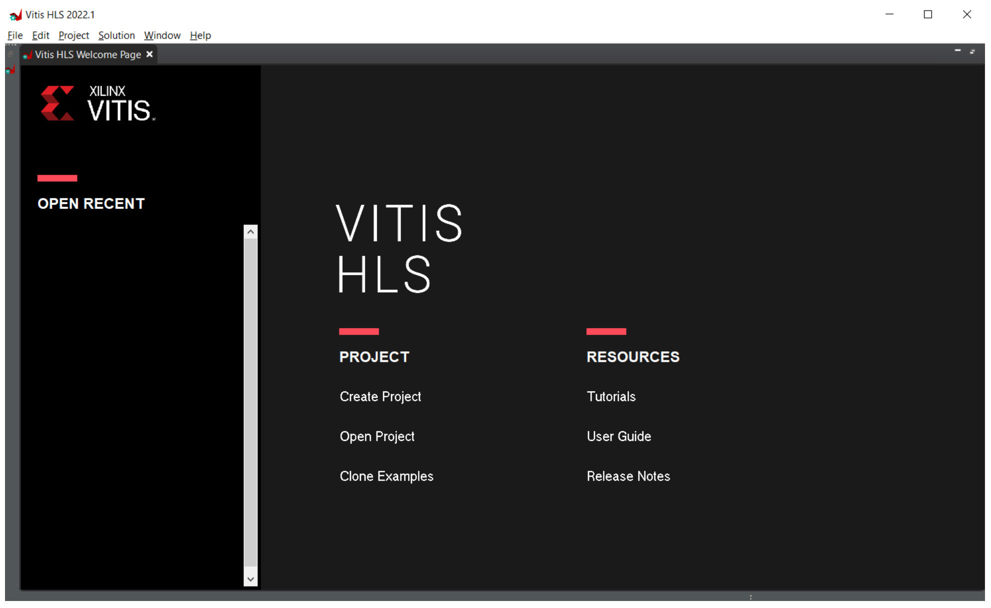


Figure 5.2: Create Project.

1. Launch `vitis_hls.bat` from **`C:\Software\Xilinx\Vitis_HLS\2022.1\bin`**.

2. In the Getting Started GUI, click on Create Project. The New Vitis HLS Project wizard opens (Figure 5.2).

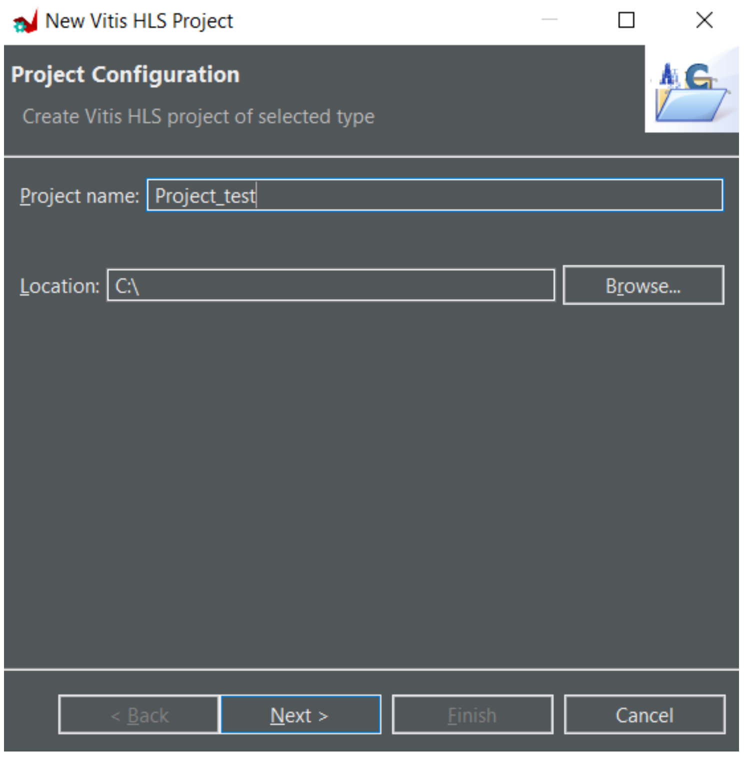
3. Chose a project name without any space or special character (Figure 5.3). It is important to note that most hardware synthesis tools and compilers do not like network units and might cause issues during simulation/synthesis. For that reason, we advise you to avoid using the N: drive and to save your project files on the local drive instead. Also, make sure to back up your files after each lab session because the lab machines frequently wipe out user-specific data on local drives.

4. Download the $import\_files.zip$ file from Learn and add all contained files to the project. Make sure to select your `top-level function` at this point and click Next.

5. Similar to the previous step, you can add your testbench here and click Next (if you do not yet have a testbench, you can simply click Next and add it later).

6. Now, you are able to configure your `Solution` settings including your desired clock of the design and its uncertainty. For this assignment, we assume that the module should run at `100Mhz` in order to communicate with other modules on the FPGA. Leave the clock uncertainty as default (1.25ns) (Figure 5.4). *It is important to note that since the chosen uncertainty is set to 1.25 ns, the design might fail to meet the timing constraints in Vitis. This is due to the fact that Vitis tries to synthesize the design with the specified clock frequency minus the uncertainty value (10ns - 1.25ns = 8.75ns). Since the IDCT component is going to be running at 100 MHz in the block design, a maximum hold slack of -1.25 ns can be tolerated in Vitis.*

7. Click on the three dots in the part selection section to choose the correct part for this project. Make sure to choose `xc7z020clg400-1` part(Figure 5.5).


New Vitis HLS Project

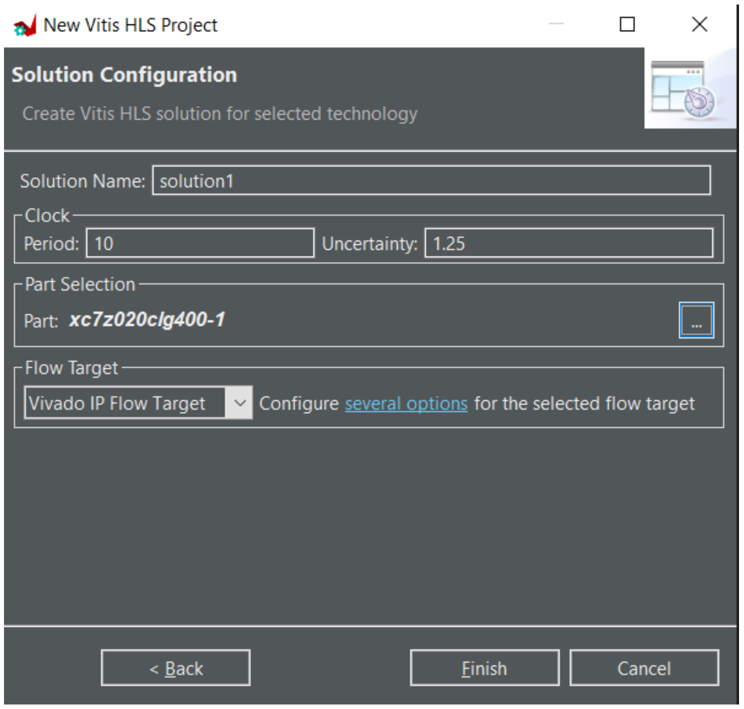
Project Configuration

Create Vitis HLS project of selected type

Project name: Project_test

Location: C:\

Figure 5.3: Project Name.


New Vitis HLS Project

Solution Configuration

Create Vitis HLS solution for selected technology

Solution Name: solution1

Clock

Period: 10 Uncertainty: 1.25

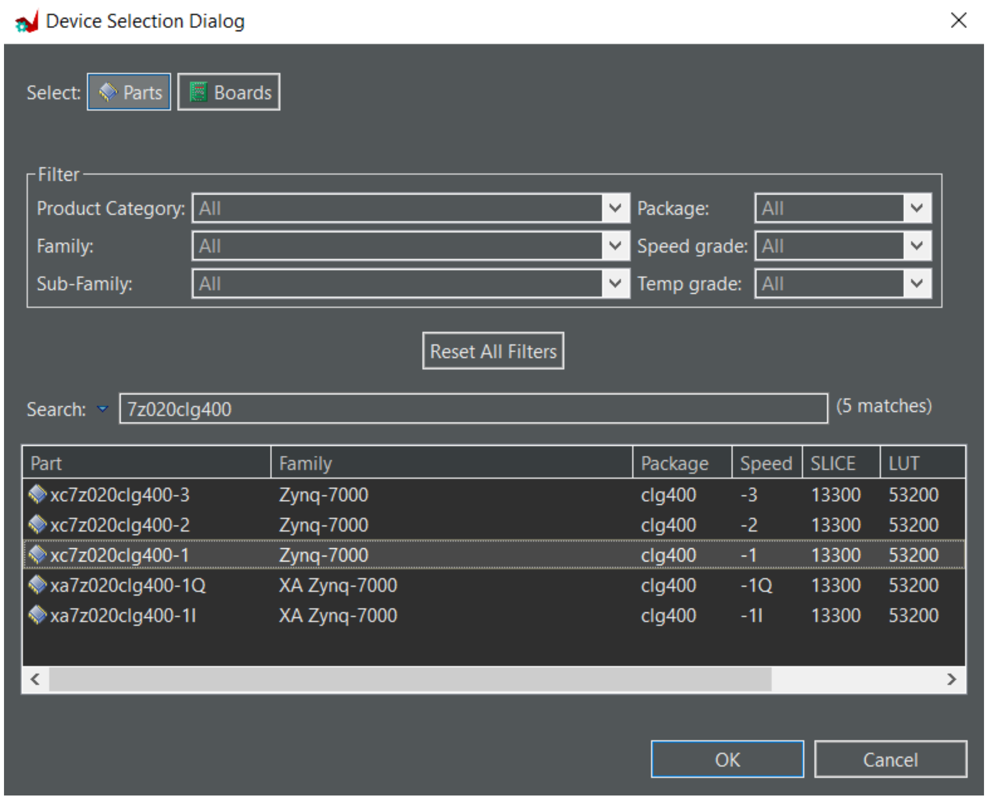
Part Selection

Part: ***xc7z020clg400-1***

Flow Target

Vivado IP Flow Target — Configure several options for the selected flow target

Figure 5.4: Solution Configuration.


Device Selection Dialog

Select: Parts Boards

Filter

Product Category: All

Family: All

Sub-Family: All

Package: All

Speed grade: All

Temp grade: All

Reset All Filters

Search: 7z020clg400 (5 matches)

Part	Family	Package	Speed	SLICE	LUT
xc7z020clg400-3	Zynq-7000	clg400	-3	13300	53200
xc7z020clg400-2	Zynq-7000	clg400	-2	13300	53200
xc7z020clg400-1	Zynq-7000	clg400	-1	13300	53200
xa7z020clg400-1Q	XA Zynq-7000	clg400	-1Q	13300	53200
xa7z020clg400-1I	XA Zynq-7000	clg400	-1I	13300	53200

OK Cancel

Figure 5.5: Part Selection.

Now, you will see the created project in the Explorer view. Expand various sub-folders to see the entries under each sub-folder. If you did not include any source file or testbench in the project, you can add them by right-clicking on `source/testbech` entries in the explorer and add a new file. In addition, the top-level function can be modified at anytime from Project → Project Setting → Synthesize.

5.2.3 Configure Interfaces

2D_idct.c is the main file that you need to start this assignment with and you can find it on Learn from $import\_files$ → 2D_idct.c. This code takes a 2 dimensional DCT block (8×8 called DCAC in the code), applies the first and second pass according to the algorithm discussed in Chapter 3, and produces a pixel color block as the output of the module. Note that the provided file has exactly the same functionality as the file with the same name contained in the sample decoder application; however, we slightly rewrote the function to make it more amenable to HLS implementation (in particular, by removing pointers).

As discussed in class, each HLS component has a control interface that determines when the IP initiates and ends transactions. For this IP, we will be using the `ap_ctrl_none` which corresponds to the implicit/none type explained in class. With this type, the HLS component starts a transaction only when input data is available. The default control interface that Vitis HLS generates is the `ap_ctrl_hs` which corresponds to the Bus (slave) type explained in class. To switch the default control interface to `ap_ctrl_none`, add the following directive at the top of your idct function: `#pragma HLS interface ap_ctrl_none port=return`

Notice that, generally, the input data that comes to the IDCT module is not a sparse matrix; therefore, you are required to read the input and generate the output **sequentially** (one row/column or one element at a time). In other words, you must configure the interface of the module such that **DCAC** and **block_out** ports are both **streaming interfaces**. In particular, the DMA core is configured to accept 32 bits streaming connections. Therefore, the DCAC and block_out ports must be 32-bit wide - this can be done by reshaping them with the correct factor/dimension.

The input to the component is 1024 bits as the DCT block (`dct_block`) and the output is 512 bits. Since the 2D-IDCT in the provided algorithm executes the columns first and then rows, the module should receive all the inputs from the streaming interface, finish the pass 1, and then start pass 2. While pass 2 is in progress and one row has been produced, they should be streamed to the output immediately.

5.2.4 C Simulation

Verification in Vitis HLS flow can be separated into two distinct processes.

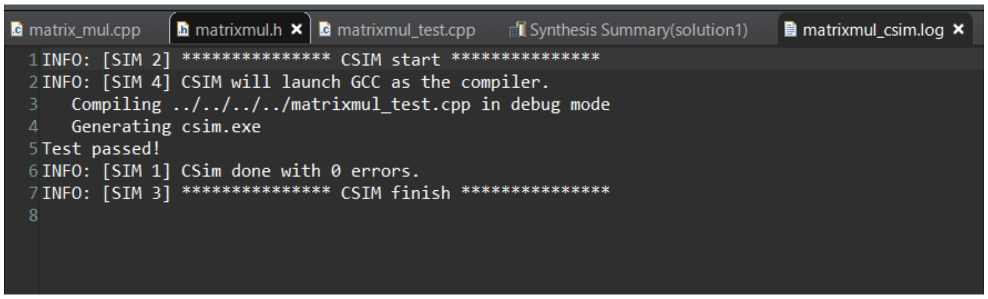
- Pre-synthesis validation that validates the C program correctness
- Post-synthesis verification that verifies the RTL correctness

Both processes are referred to as simulation: C simulation and C/RTL co-simulation. Before synthesis any design, the function to be synthesized should be validated with a testbench using C simulation. A C testbench includes a top-level function main() and the function to be synthesized. An ideal testbench has the following attributes:

- The testbench is self-checking and verifies the results from the function to be synthesized
- If the results are correct, the testbench returns a value of 0; otherwise, the testbench should return any non-zero value

Note that we provide on Learn two 8x8 block examples (input coefficients and resulting output color values) that you can use to write your testbench.

Clicking the Run C Simulation toolbar button opens the C simulation dialog box and the C simulation is automatically executed. Notice that you can debug your code by choosing the Launch Debugger in the C simulation dialog box. When C simulation completes, a folder `csim` will be created inside the solution and a log file is generated and placed in `csim/report`. A sample output for the C simulation is shown in Figure 5.6

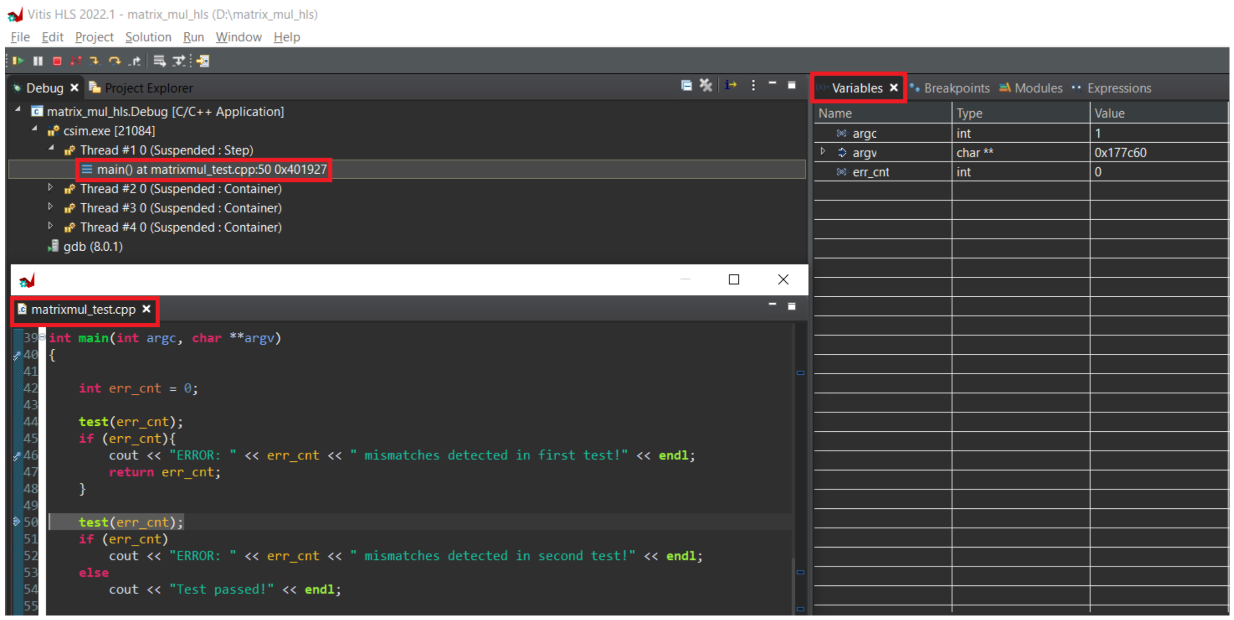


```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../matrixmul_test.cpp in debug mode
   Generating csim.exe
Test passed!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figure 5.6: C Simulation Output.

5.2.5 Debugger Mode

Debugger mode helps to understand the behavior of the program. Select Project → Run C Simulation or click on its symbol from the tools bar buttons. Select the Launch Debugger option and click OK. The Debug perspective will show the source code in the source view, and variables defined in the Variables view, Outline view showing the objects which are in the current scope, thread created and the program suspended at the main() function entry point. A sample Debug mode is shown in Figure 5.7. You can use breakpoints, step over, step into, step over, and resume the code functions to leverage different debugging methods.


Vitis HLS 2022.1 - matrix_mul_hls (D:\matrix_mul_hls)
File Edit Project Solution Run Window Help
Debug
Project Explorer
Variables
Breakpoints
Modules
Expressions
matrixmul_test.cpp

```
matrix_mul_hls.Debug [C/C++ Application]
  csim.exe [21084]
    Thread #1 0 (Suspended : Step)
      main() at matrixmul_test.cpp:50 0x401927
    Thread #2 0 (Suspended : Container)
    Thread #3 0 (Suspended : Container)
    Thread #4 0 (Suspended : Container)
  gdb (8.0.1)
```

Name	Type	Value
argc	int	1
argv	char **	0x177c60
err_cnt	int	0

```
int main(int argc, char **argv)
{

    int err_cnt = 0;

    test(err_cnt);
    if (err_cnt){
        cout << "ERROR: " << err_cnt << " mismatches detected in first test!" << endl;
        return err_cnt;
    }

    test(err_cnt);
    if (err_cnt)
        cout << "ERROR: " << err_cnt << " mismatches detected in second test!" << endl;
    else
        cout << "Test passed!" << endl;

```

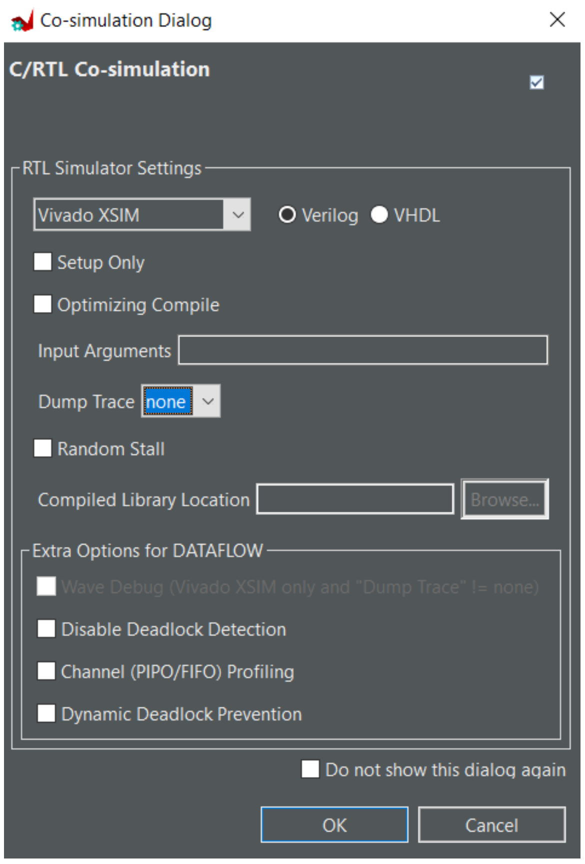
Figure 5.7: Debug Mode Overview.

5.2.6 Synthesize and Extract the Performance

Once you are done implementing and testing your 2D IDCT, you are ready to execute the synthesize. Set **100 Mhz** as the frequency of the system. Use the C Synthesis toolbar button or navigate from Solution → Run C Synthesize to synthesize the design to an RTL implementation. During the synthesis process messages are echoed to the console window. When synthesis completes, the synthesis report for the top-level function opens automatically in the information panel and the folder `syn` is now available in the solution folder. The `syn` folder contains four sub-folders: a report folder and one folder for each of the RTL output formats. The report folder contains a report file for the top-level function. This report includes timing and resource utilization information.

5.2.7 C/RTL Co-Simulation

Post-synthesis verification is required to guarantee the correctness of the generated RTL by the tool. Click on Solution → Run C/RTL co-simulation to verify the RTL results. A C/RTL co-simulation dialog box will open. The default option for RTL Co-simulation is to perform the simulation using the built-in C compiler and verify the results by passing them to the RTL design. This type of simulation will be used to test the correctness of the IDCT output values. The C/RTL co-simulation dialog box allows you to select which type of RTL output to use


Co-simulation Dialog

C/RTL Co-simulation

RTL Simulator Settings

Vivado XSIM   Verilog [✓ VHDL]

Setup Only

Optimizing Compile

Input Arguments

Dump Trace: none

Random Stall

Compiled Library Location

Extra Options for DATAFLOW

Wave Debug (Vivado XSIM only and "Dump Trace" != none)

Disable Deadlock Detection

Channel (PIPO/FIFO) Profiling

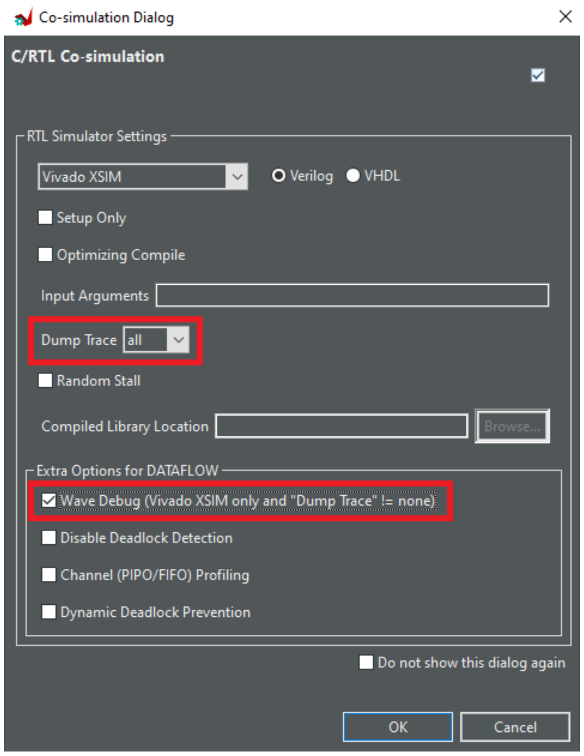
Dynamic Deadlock Prevention

Do not show this dialog again

Figure 5.8: Co-simulation Dialog

for verification (Verilog or VHDL) and which HDL simulator to use for the simulation (leave the default options) as can be seen in Figure 5.8. RTL co-simulation only reports a successful verification when the test bench returns a value of 0 (zero).

In addition to checking the correctness of your output values, it is also important to check if the streaming interface of your HLS component is outputting the datasets in the correct order. For that, you have to check the generated waveform, and make sure that the ordering of the input and output is correct. You can view the waveform of your testbench by setting the dump trace dropdown to all and by checking the Wave Debug checkbox as shown in Figure 5.9. The generated waveform (.wdb) file will be generated in `<project_directory>/solution1/sim/verilog/`**`project_name.wdb`**



Co-simulation Dialog

C/RTL Co-simulation

RTL Simulator Settings

Vivado XSIM ○ Verilog ● VHDL

☐ Setup Only

☐ Optimizing Compile

Input Arguments

Dump Trace: all

☐ Random Stall

Compiled Library Location [Browse...]

Extra Options for DATAFLOW

☑ Wave Debug (Vivado XSIM only and "Dump Trace" != none)

☐ Disable Deadlock Detection

☐ Channel (PIPO/FIFO) Profiling

☐ Dynamic Deadlock Prevention

☐ Do not show this dialog again

OK Cancel

Figure 5.9: Generating input and output waveforms of the HLS component

5.2.8 Optimization

High-level synthesis creates an optimized implementation based on default behavior, constraints, and any optimization directives you specify. You can use optimization directives to modify and control the default behavior of the internal logic. This allows you to generate **variations** of the hardware implementation from the **same** C/C++ code. To determine if the design meets your requirements, you can review the performance metrics in the synthesis report generated by high-level synthesis. After analyzing the report, you can use optimization directives to refine the implementation. Here are some examples of the optimizations that can be applied in HLS directives/pragmas:

- **Pipelining**: enables an iteration of a function or a loop to be executed before the previous one is over. It increases throughput with minimal resource usage increase.
- **Unrolling**: enables multiple iterations of a for loop to run in parallel, if independent. Greatly reduces latency and have an impact on resource usage based on loop size.
- **Dataflow**: enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the

overall throughput of the design. Notice that if you employ Dataflow optimization, you need to test the design with two transactions since otherwise the C/RTL co-simulation can not check the interval.

Most of the required optimizations to achieve the desired throughput for this assignment are discussed in the class through matrix multiplication example and demonstrated in the HLS assignment tutorial. For further HLS directives references and usage see Xilinx HLS Directive or go to the next url: `https://docs.xilinx.com/r/2022.1-English/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS`. Note that the 2D-IDCT consists of two loops, which are both potential targets for optimizations. After optimizing the design, make sure to run synthesis, C simulation and C/RTL co-simulation again. **Note:** Vitis HLS automatically tries to apply some optimizations, in particular loop pipelining. However, you can still specify your desired optimizations through pragmas, and in particular, you might need to partition / resize data structures as needed.

In this task, your final goal is to optimize the IP using the optimization directives, with the objective to achieve minimum possible interval in clock cycles for the module (still at 100 Mhz) without using unnecessary hardware resources. Notice that, as long as the minimum interval is achieved, the latency in clock cycles of your design is not important. **Hint:** apart from processing, the accelerator must also transfer inputs and outputs; such transfer times are constrained by the specified width of the streaming interfaces. Clearly, the interval cannot be shorter than the time required to read the inputs or write the outputs, hence there is no point trying to make processing faster than I/O time.

Note: adding optimization directives might make it harder for the tool to satisfy the 10ns timing constraint. If the tool fails by returning a negative slack (make sure to check the synthesis results), you can try to synthesize the design again after setting a period lower than 10ns. This can cause the tool to increase the number of pipeline stages for each loop, in turn reducing the length of the critical path and thus meeting the original constraint (e.g., if you set a period of 8ns and you obtain a slack of -1.5ns, then you know that the critical path is 9.5ns $<$ 10ns).

5.3 Task 2: IP Implementation and Integration

5.3.1 Exporting the IDCT Accelerator as an IP Block

After optimizing and verifying that your IDCT Accelerator is behaving in the correct manner, the HLS component must be exported as an IP block and integrated with the prefab block diagram. Please follow the steps below to export the IDCT Accelerator as an IP block:

- click on `Solution ... Export RTL` as shown in Figure 5.10

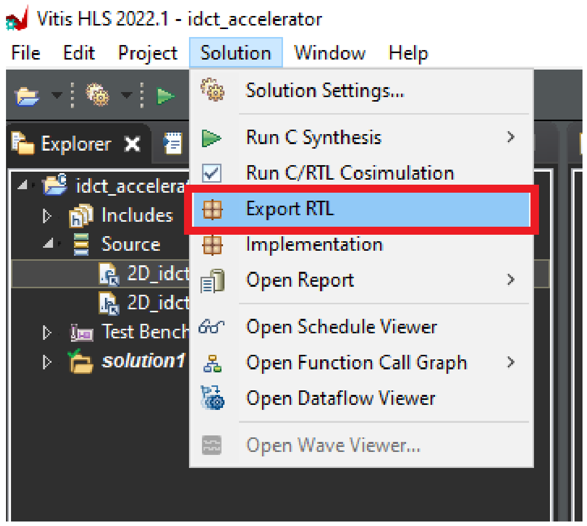
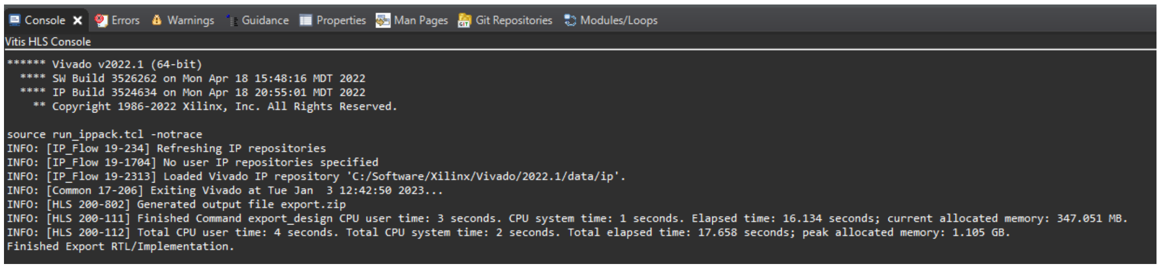


Figure 5.10: Export RTL

- In the `Export Format` select `Vivado IP (.zip)`. Also, select the output location of the IP block (Leave all the other fields empty).
- Upon completion you should see in the console the message displayed in Figure 5.11



```
****** Vivado v2022.1 (64-bit)
  **** SW Build 3526262 on Mon Apr 18 15:48:16 MDT 2022
  **** IP Build 3524634 on Mon Apr 18 20:55:01 MDT 2022
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'C:/Software/Xilinx/Vivado/2022.1/data/ip'.
INFO: [Common 17-206] Exiting Vivado at Tue Jan  3 12:42:50 2023...
INFO: [HLS 200-802] Generated output file export.zip
INFO: [HLS 200-111] Finished Command export_design CPU user time: 3 seconds. CPU system time: 1 seconds. Elapsed time: 16.134 seconds; current allocated memory: 347.051 MB.
INFO: [HLS 200-112] Total CPU user time: 4 seconds. Total CPU system time: 2 seconds. Total elapsed time: 17.658 seconds; peak allocated memory: 1.105 GB.
Finished Export RTL/Implementation.
```

Figure 5.11: Console message when Vitis HLS finishes generating the IP block

- Go to the output folder and extract the generated `export.zip` file.
- Switch to the `lab_prefab` Vivado project. Select `Tools ... Settings`. In `Project Settings` expand IP and select `Repository`. Click on the plus icon and select the extracted IP block directory as shown in Figure 5.12.

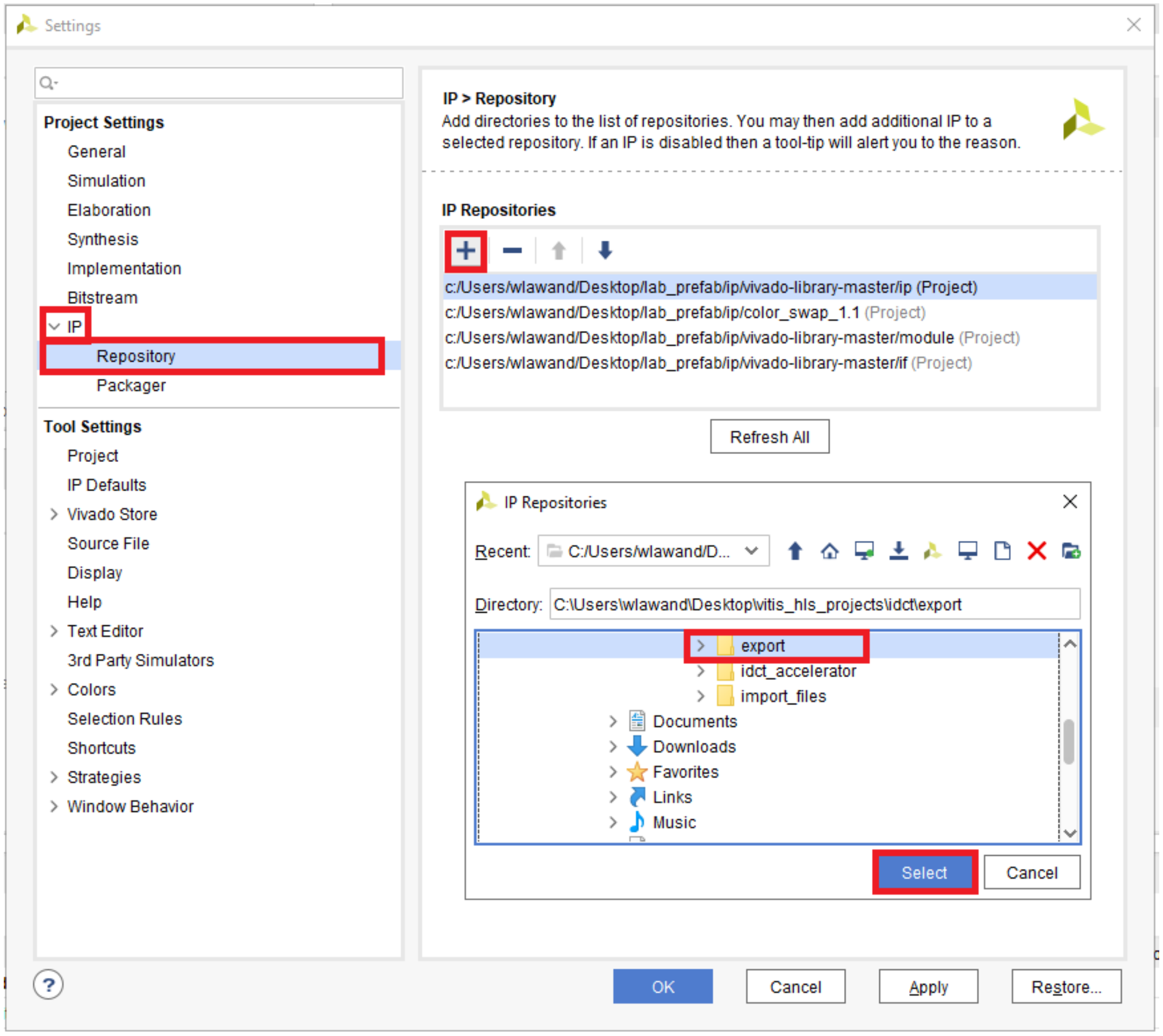
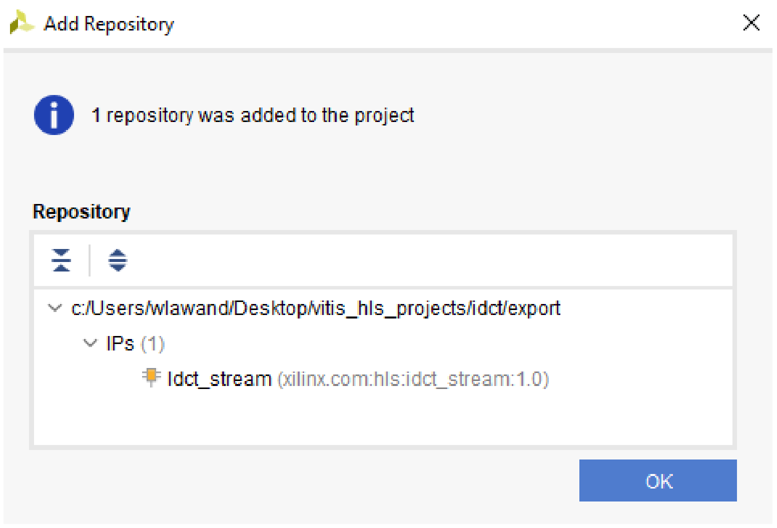

Settings
Project Settings
General
Simulation
Elaboration
Synthesis
Implementation
Bitstream
IP
Repository
Packager
Tool Settings
Project
IP Defaults
Vivado Store
Source File
Display
Help
Text Editor
3rd Party Simulators
Colors
Selection Rules
Shortcuts
Strategies
Window Behavior
IP > Repository
Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.
IP Repositories
c:/Users/wlawand/Desktop/lab_prefab/ip/vivado-library-master/ip (Project)
c:/Users/wlawand/Desktop/lab_prefab/ip/color_swap_1.1 (Project)
c:/Users/wlawand/Desktop/lab_prefab/ip/vivado-library-master/module (Project)
c:/Users/wlawand/Desktop/lab_prefab/ip/vivado-library-master/if (Project)
Refresh All
IP Repositories
Recent: C:/Users/wlawand/D...
Directory: C:\Users\wlawand\Desktop\vitis_hls_projects\idct\export
export
idct_accelerator
import_files
Documents
Downloads
Favorites
Links
Music
Select
Cancel
OK
Cancel
Apply
Restore...

Figure 5.12: Linking the Vivado project to the generated IP

- After selecting the correct directory a dialog similar to the one in Figure 5.13 showing the name of the added IP must appear.


Add Repository

1 repository was added to the project

Repository

c:/Users/wlawand/Desktop/vitis_hls_projects/idct/export
IPs (1)
Idct_stream (xilinx.com:hls:idct_stream:1.0)

Figure 5.13: Added IPs dialog box

- At this point, you should be able to add the IDCT Accelerator from the IP catalog by pressing `ctrl+I` and searching for the name of the IP as shown in Figure 5.14.
- Connect the `DCAC` and `blockout` ports of the imported accelerator to the `M_AXIS_MM2S` and `S_AXIS_S2MM` ports of the DMA respectively. To do so, you first need to delete the loop connection between `M_AXIS_MM2S` and `S_AXIS_S2MM` ports.

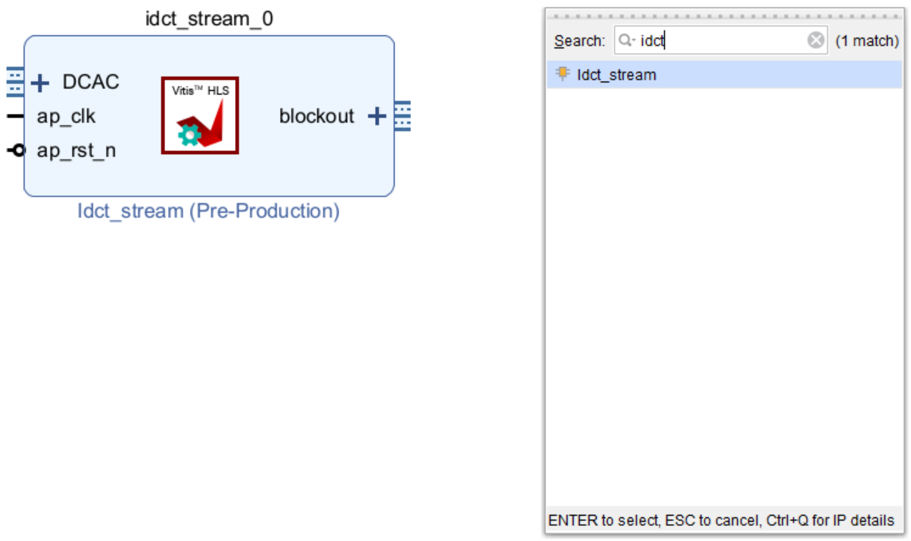

idct_stream_0
DCAC
ap_clk
ap_rst_n
Vitis™ HLS
blockout
Idct_stream (Pre-Production)
Search: idct
(1 match)
Idct_stream
ENTER to select, ESC to cancel, Ctrl+Q for IP details

Figure 5.14: IP added to the Block Design

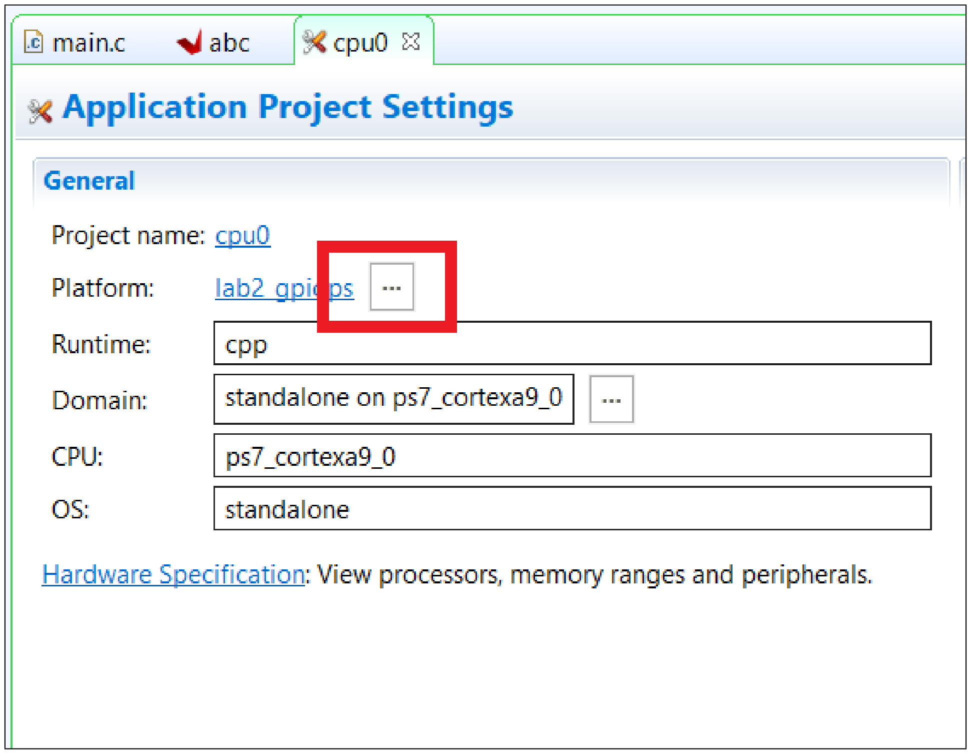
- After connecting those ports, a green box at the top of the screen will pop up, prompting to `Run Connection Automation`. Click on the hyperlink and make sure that 100 MHz is selected as the clock frequency. After pressing OK, you should find that the `ap_clk` port of the accelerator got connected to `FCLK_CLK_0` that is running at 100 MHz.

5.3.2 Using the IDCT Accelerator in the Decoder Application

After you integrate the Accelerator in Vivado, make sure to regenerate the bitstream and hardware definition file as explained in Assignment 1. Then in Vitis, follow the steps below:

1. Go to file ->New ->Platform Project... Type in the name for the new project. Click Next and select the new .xsa file. And click Finish.

2. Go to Application Project Settings (.prj file as mentioned in Assignment 1). Click on the button next to Platform. In the window that pops up, select the newly created platform and click OK. Click Yes to clean the build configurations.


main.c
abc
cpu0
Application Project Settings
General
Hardware Specification: View processors, memory ranges and peripherals.

Project name:	cpu0
Platform:	lab2_gpio_ps
Runtime:	cpp
Domain:	standalone on ps7_cortexa9_0
CPU:	ps7_cortexa9_0
OS:	standalone

Figure 5.15: Building the project

3. Make sure to rebuild the project in desired Build Configuration (Debug or Release).

4. Note that you also need to create a new Run Configuration with the new bitfile. To delete previous configuration: Right Click ->Delete. To create new Run Configuration, follow the Step No. 6 from the Section 4.4 from Assignment 1.

Finally, modify your software application to use the hardware accelerator. Specifically, configure the DMA to read from memory the `YDCAC`, `CbDCAC` and `CrDCAC` matrixes for the entire frame, and to write to memory the corresponding pixel color blocks. Make sure to manage cache coherence as required.

5.4 Task 3: Profile the Application

Profile again the application in the same way as in Assignment 1. You must report the new time for the IDCT functionality. Also report the time required to flush / invalidate each cache level. **Note:** the time required to perform the IDCT should be roughly equal to the number of processed 8x8 blocks times the interval of the accelerator times 10ns. Make sure this is the case (in general, you should always run sanity checks on the timing of the various modules after every optimization).

5.5 Deliverables

Demo: during the demo, you need to demonstrate the ability to decode the video using the designed hardware accelerator. Similar to Assignment 1, you must first demonstrate the application without the periodic timer (i.e., decoding each frame one after the other), and then using the timer in conjunction with playback control. Be ready to show your accelerator implementation under Vitis HLS and the results of synthesis and simulation. You must submit to learn the following files for the accelerator after optimization: (1) the modified 2D_idct.c file, including all applied optimization directives; (2) the testbench used for simulation; (3) the report file (.rpt) generated by synthesis; (4) the log file (.log) created by C simulation; (5) the waveform (.wdb) generated by co-simulation. Also submit to Learn all modified files in the decoder application.

Report: your report must include: (1) a copy of the "Performance and Resource Estimate" table from the synthesis report; (2) a table reporting the required profiling information. A template describing the required format for the report is provided on Learn.

The mark breakdown is reported in Table 5.1. For the demo component, you will be marked based on correctness. For the optimization component, you will be marked on a scale based on your ability to meet the stated objective (minimize the interval without using unnecessary hardware resources). **Note:** we are not telling you what is the optimal result because in a realistic industrial project, there would be no way of knowing it. Once you believe your design to be sufficiently optimized, do not spend an unbounded amount of time trying to shave off the last clock cycle or LUT - it is not time well spent - all projects that we consider to be well-optimized will receive the maximum grade. However, keep into mind that as mentioned in Section 5.4, the interval determines the execution time of the IDCT. The shorter such time is, the easier it will be to derive a schedule in Assignment 3. In particular, the time required to perform the IDCT for all three color components of a frame should be (well) below 41ms, since otherwise there will be no way to meet the 24fps constraint. Hence, make sure you spend at least a **reasonable** optimization effort on minimizing the interval.

Total Assignment 2	8%
Demo	4%
Optimization	3%
Report	1%

Table 5.1: Mark Breakdown For Assignment 2

Chapter 6

Assignment 3

In Assignment 3, you will design a parallel schedule for the MJPEG423 decoder application and implement it on the Zynq embedded board. Section 6.2 details how to run a multicore application on the board, while Section 6.3 presents the task for this assignment.

6.1 Task 1: Design the Application Schedule

Your goal of this task is to produce an optimized periodic schedule for the MJPEG423 decoder application, following the techniques discussed in class for task graph scheduling. Your schedule must respect the following set of constraints.

- The platform comprises one hw accelerator used to execute the IDCT and 2 ARM Cortex A9 cores.
- Since you must display frames periodically at 24 fps, the application has an output periodicity of $1000/24 = 41.67$ ms. There is no required input periodicity, since you can read the video files whenever you want.
- The worst-case repeating sequence of I and P-frames encountered in any video corresponds of I-frame, I-frame, 23 P-frames; you should perform relevant period and buffer size math based on such pattern. However, at run-time you must be able to adjust your schedule by inserting a variable number (0 to 23) of P-frames between two I-frames, since in reality, such number varies during the video.
- Makespan is not an important metric - this is video playback, not videoconferencing - so do not spend effort minimizing it. Instead, try to find the easiest (thus less work required) repeating schedule that fits the constraints. Specifically, it is ok to perform the

math assuming a shorter repeating sequence - that is, if it simplifies your design, you can assume a sequence of I-frame, I-frame, x P-frames, where $x < 23$ (however, the smaller x is, the harder it will be to meet the period constraints, given that I-frames are more computationally intensive than P-frames).

- You need to include tasks to perform the following 6 functional components: (1) SD Read: one task per frame; it reads a file portion from SD card and outputs a Y' bitstream, a CR bistream and a CB bitstream; (2) lossless decode; (3) hw IDCT; (4) DMA programming: used to either start DMA transfers for the accelerator, or check that transfers have completed; (5) color conversion; (6) any required cache flushing/invalidation operation (specify the cache API function(s) being used). Note that as explained in Section 5.1.2, you must model the buff_reg function as a cache flushing operation since it flushes L1 and L2. You do not need to model the video IP output.

- To determine the time to execute each functional component, use your profiling results from Assignment 1 and 2. To be on the safer side, we suggest using worst-case timing values for I-frames, and average case values for P-frames (this is because the repeating sequence includes only 2 I-frames, but 23 P-frames). Assume that the time for DMA programming is zero. You can create a task by processing any number of blocks, as long as such number is at least 160 (i.e., a whole row of blocks; below this you would be adding too much synchronization overhead to the implementation and the numbers would not be correct anymore). The execution time of the task should be proportional to the number pf processed blocks. **Example:** if you want to balance the color conversion load among two cores, you could create two color conversion tasks processing half the color blocks each; the time for each task would then be half the profiled time for color conversion.

- Once you have determined your tasks, derive a DAG based on the precedence constraints in Figures 3.4, 3.5. Note that based on the precedence constraints in the figures, IDCT and color conversion tasks can be executed in parallel, but lossless decode (for the same frame and color component) cannot. Also note that based on Figure 3.5, you cannot start executing the lossless decode on a given block for a P-frame until you have completed the IDCT on that same block for the previous frame. This is because the lossless decode for P-frames overwrites the content of the corresponding DCAC data structure; hence, if you execute it before the IDCT of the previous frame, you would incorrectly modify the IDCT input.

- The figures do not include the SD Read task, nor the DMA programming tasks nor any cache management task. Since all lossless_decode tasks require a bitstream as input, you should assume that they depend on the SD Read for that frame. In the case of the DMA programming task, simply replace the IDCT task with a chain of DMA programming $\rightarrow$ IDCT $\rightarrow$ DMA programming (i.e., the first DMA programming task - to start the DMA transfer - must follow the lossless_decode, and the second task - to check that the transfer has completed - must precede the color conversion). A similar strategy can be used for the cache management tasks.

- Regarding cache management: when adding such tasks, keep in mind that the platform has private L1 caches and a shared L2 cache. This means that flushing/invalidating L2 affects both cores; but calling the L1 flush/invalidate function from a core only affects the cache content in the L1 of that specific core.

- Because of programming restrictions, you must respect the following additional constraints: (1) all instances of the SD Read task must be executed on the same core. (2) All instances of DMA programming tasks must be executed on the same core (but it does not need to be the same core as the SD Read). (3) All calls to the video library must be made on the same core (again, it doesn't have to be the same core as SD Read and DMA programming). **Note: this is the reason we must model DMA programming even if we ignore its computation time.** Specifically, remember that in our model tasks must run to completion once started; hence, even if a DMA programming task takes zero time, you cannot interrupt the execution of another task on the chosen core to execute it.

  In addition to the schedule, you must compute bounds on the size in bytes of all required buffers. Again, use the profiling information from Assignment 1 to determine the size of each data structure (i.e., each token in the FIFO buffer in dataflow terminology). Note that the size of the framebuffer must be increased by one since the video display API must retain the currently displayed frame in the buffer, while the output buffer computation used in class assumes that the data is consumed (removed from buffer) immediately when the periodic timer handler is executed.

6.2 Multicore Design

We start by discussing how to synchronize the two ARM Cortex A9 cores in Section 6.2.1. Then, we discuss how to create and execute a software application in Sections 6.2.2 - 6.2.4.

6.2.1 Synchronization Primitives

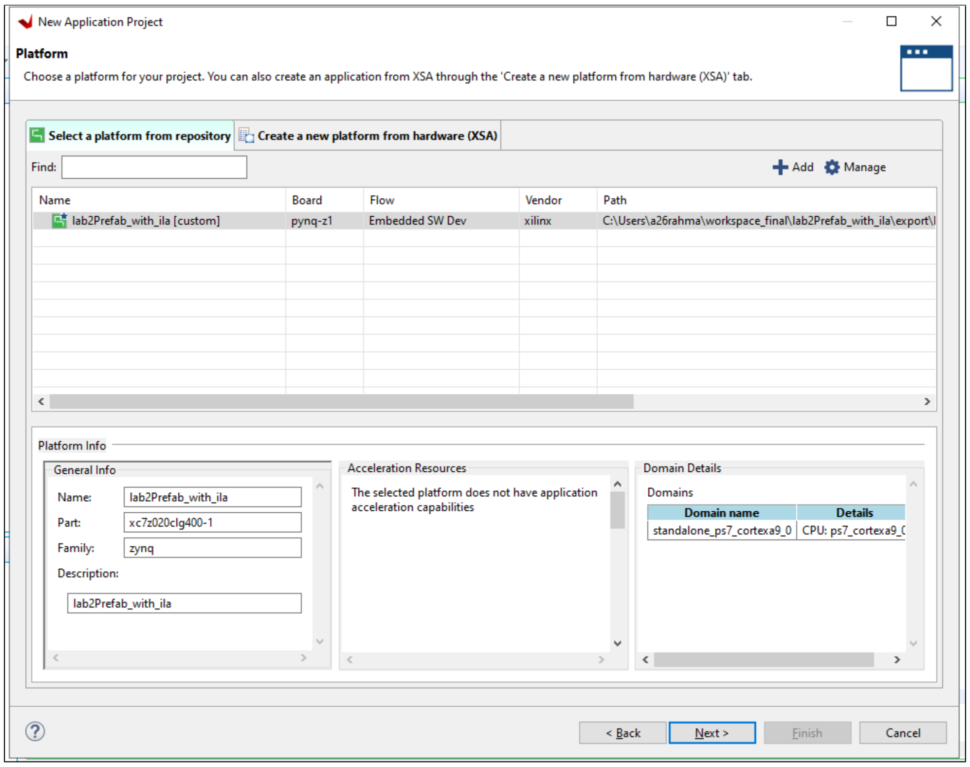
In the SystemC simulation, cores were synchronized using FIFO queues. In your implementation, you should replace the relevant queues with circular buffers; note that you can determine the position in the circular buffer you should read / write into based on the frame number. There is no memory isolation between the cores, in the sense that each core can read/write data structures allocated by the other core.

To support inter-core synchronization, we suggest to employ ARM spinlocks; refer to files spinlock.c and spinlock.h provided on Learn. Calling spin_lock on an integer variable with value 0 causes the core to block by entering standby mode. Calling spin_unlock on an integer variable sets it to 1 and unblocks any core in standby mode.

6.2.2 Create a Multicore Application

After you have followed steps till Assignment 2, where you have a hardware platform with accelerator in your workspace, and a single core application, you can proceed on to create another application for the other core with the same hardware platform. To create and run the application on the second core, you do the following:

1. Make sure you are in your previous workspace from Assignment 2
2. Go to File ->New ->Application Project. In the popup window that opens, click Next. The hardware platform you created in Assignment 2 should be available in the "Select a platform from repository" tab. Select that and click next.



Platform

Choose a platform for your project. You can also create an application from XSA through the 'Create a new platform from hardware (XSA)' tab.

Find:

Name	Board	Flow	Vendor	Path
lab2Prefab_with_ila [custom]	pynq-z1	Embedded SW Dev	xilinx	C:\Users\a26rahma\workspace_final\lab2Prefab_with_ila\export\l

Name: lab2Prefab_with_ila
Part: xc7z020clg400-1
Family: zynq
Description: lab2Prefab_with_ila

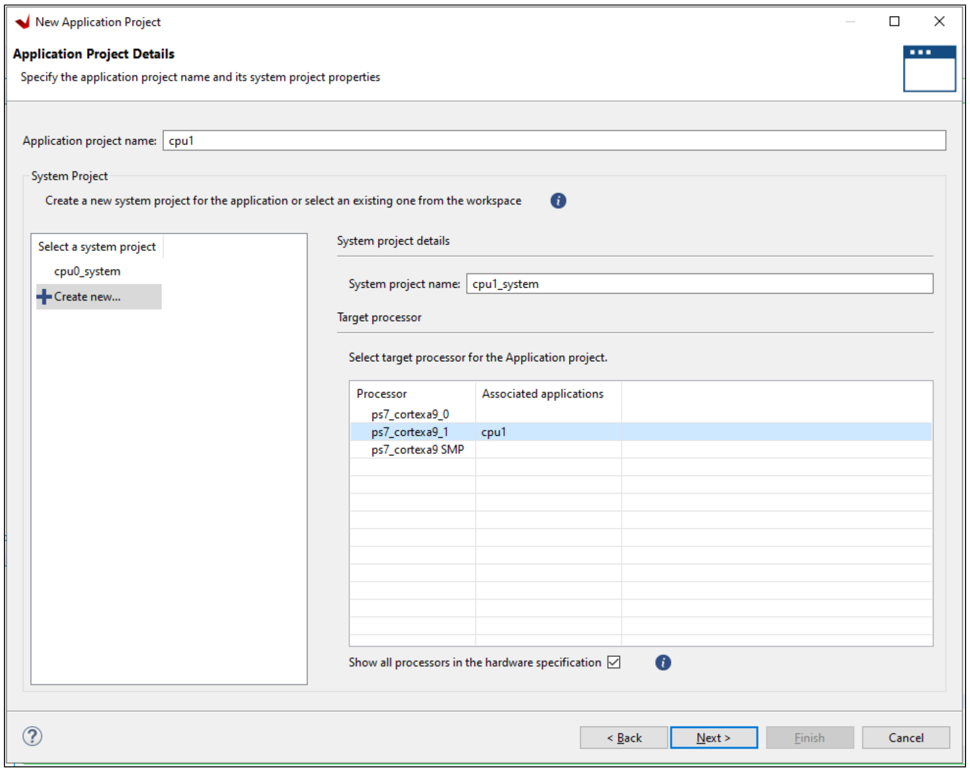
The selected platform does not have application acceleration capabilities

Domains

Domain name	Details
standalone_ps7_cortexa9_0	CPU: ps7_cortexa9_(

Figure 6.1: Building the project

3. In the Application Project Details pane, select the second processor (named ps7_cortexa9_-1). In case this is not visible, check the box to "Show all processors in the hardware specification".


New Application Project

Application Project Details

Specify the application project name and its system project properties

Application project name: cpu1

System Project

Create a new system project for the application or select an existing one from the workspace

Select a system project
- cpu0_system
- Create new...

System project details

System project name: cpu1_system

Target processor

Select target processor for the Application project.

Processor	Associated applications
ps7_cortexa9_0	
ps7_cortexa9_1	cpu1
ps7_cortexa9 SMP	

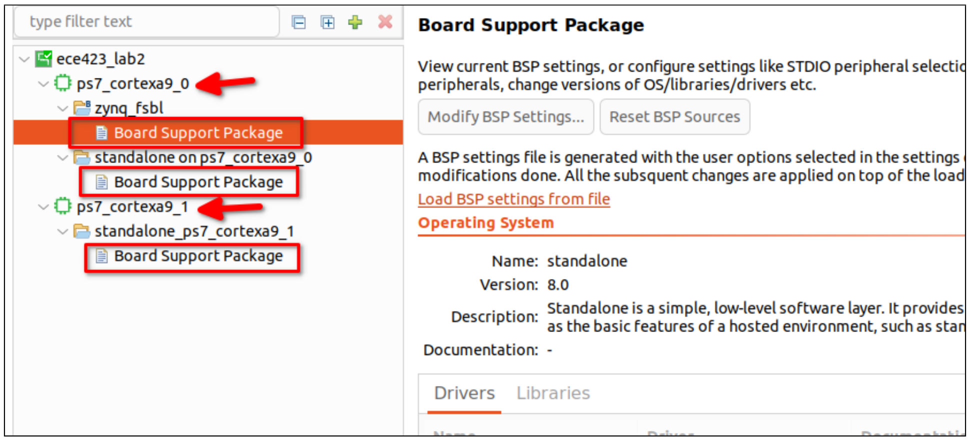
Show all processors in the hardware specification ☑

Figure 6.2: Building the project

4. Click Next ->Next ->and choose if you want to use a template and finish creating the application project.

5. It is to be noted that you should use the hardware resources (DMA, IDCT, Video, SD Card) only on single core in order to avoid hardware access conflicts. Prefer using UART on single core as well. But all of these do not necessarily have to be on the same core. Furthermore, you should include the SD card library and driver (xilffs Refer to Appendix A.1) only on one of the cores.

6. Open the platform.spr file. Now here, on the left pane, you should see the secondary core: ps7_cortexa9_1 appear. Now you have to add some compilation flags to enable proper master-slave (ps7_cortexa9_0 is master and ps7_cortexa9_1 is slave) configuration. To do this, you have to select "Board Support Package" in all three suboptions and select "Modify BSP Settings" for each.


type filter text

- ece423_lab2
  - ps7_cortexa9_0
    - zynq_fsbl
      - Board Support Package
    - standalone on ps7_cortexa9_0
      - Board Support Package
  - ps7_cortexa9_1
    - standalone_ps7_cortexa9_1
      - Board Support Package

Board Support Package

View current BSP settings, or configure settings like STDIO peripheral selecti
peripherals, change versions of OS/libraries/drivers etc.

Modify BSP Settings... | Reset BSP Sources

A BSP settings file is generated with the user options selected in the settings
modifications done. All the subsequent changes are applied on top of the load

Load BSP settings from file

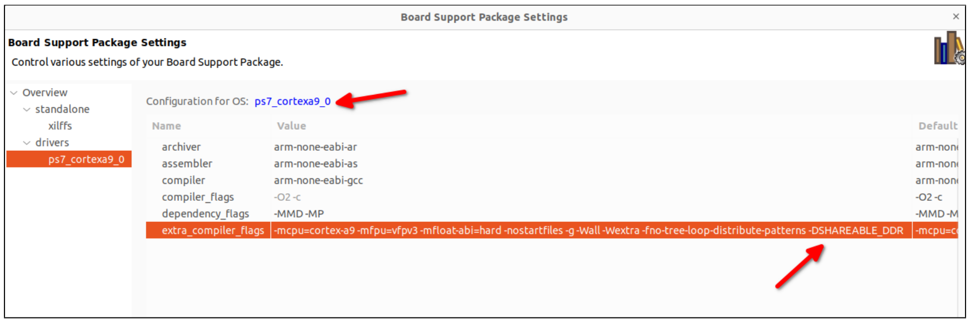
Operating System

Name:	standalone
Version:	8.0
Description:	Standalone is a simple, low-level software layer. It provides as the basic features of a hosted environment, such as star
Documentation:	-

Drivers | Libraries

Figure 6.3: Modifying BSP settings

7. For the BSP settings under ps7_cortex_a9_0 you have to go Overview ->drivers ->ps7_cortexa9_0 and double click on the "Value" column beside extra_compiler_flags. There you should get the option to edit. Leaving everything same, at the end, give a space and add: "-DSHAREABLE_DDR". Do this for both BSP settings highlighted ybder ps7_cortexa9_0 6.3. For ps7_cortexa9_1, you have to add "-DSHAREABLE_DDR -DUSE_AMP=1". These flags are passed to the compiler.


Board Support Package Settings

Board Support Package Settings

Control various settings of your Board Support Package.

- Overview
  - standalone
    - xilffs
  - drivers
    - ps7_cortexa9_0

Configuration for OS: ps7_cortexa9_0

Name	Value	Default
archiver	arm-none-eabi-ar	arm-non
assembler	arm-none-eabi-as	arm-non
compiler	arm-none-eabi-gcc	arm-non
compiler_flags	-O2 -c	-O2 -c
dependency_flags	-MMD -MP	-MMD -M
extra_compiler_flags	-mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard -nostartfiles -g -Wall -Wextra -fno-tree-loop-distribute-patterns -DSHAREABLE_DDR	-mcpu=c

Figure 6.4: Modifying Flags

6.2.3 Multicore Memory Map

The project employs a 512 MBytes DDR3 RAM for data. When implementing a multicore design, both memories must be partitioned among the ARM Cortex A9 cores. In a typical system, the most complex step involves correct configuration of the linker on each core, but luckily, Vitis simplifies this process. Following steps explain how to change the memory configuration for the application to get required memory for both cores.

In each of your application projects, you would have a linker script (lscript.ld) file in src folder. You can use that to change the memory map configuration. You can manipulate the base address and size of ps7_ddr_0 to specify how much memory is allocated to each core. Based on size from the first core, you can calculate the starting address of ps7_ddr_0 for the second core and adjust it's size as well. (Warning: the memory regions of both cores should not overlap or it may cause unpredictable issues). The sizes should at least be enough to allow stack, heap and code to be placed.

You will also need to adjust the Stack size and Heap size. Stack size is to be adjusted to a suitable value to allow enough space for local variables, function call frames etc. Heap size should be large enough to allow malloc'ing all required data structures on that core; you should know the required size for each data structure based on the profiling in Assignment 1 and the resulting buffer computation in Task 1.

Finally, you also need to leave a memory block to allow space for shared data; the simpler way to do that is to leave a chunk of memory which is neither included in memory regions of first core, nor the second core. The reason this is needed is because there is no simple way for a core to know the memory addresses at which local, global and heap variables are allocated by the other core. Hence, you should manually allocate required variables (such as the spinlock variables) in the shared block and hardcode their addresses in both applications.

For your master core (ie, one which you developed your application initially) you have to add the following block of code in your lscript.ld file. You can access the code by opening the lscript.ld file and clicking source from bottom left corner. Add the given listing at end after `_end = .;`. Make sure to replace the address by your chosen shared address. Any spinlocks you use should be pointers to this section of memory. This address should preferably reside after your allocation of ram for both the cores. This address should also be aligned to 1MB.

```
  . spinlock_section  <Add your shared address here>: {
KEEP(*(.spinlock_section))
FILL(0)
. += 0x1000;
. = ALIGN(0x100000);
  }
```

After you have done this, add the following global variable in your main file for the master core: `int dummyvar __attribute((section(".spinlock_section")));`. This makes sure that spinlock memory is initialized with zeros.

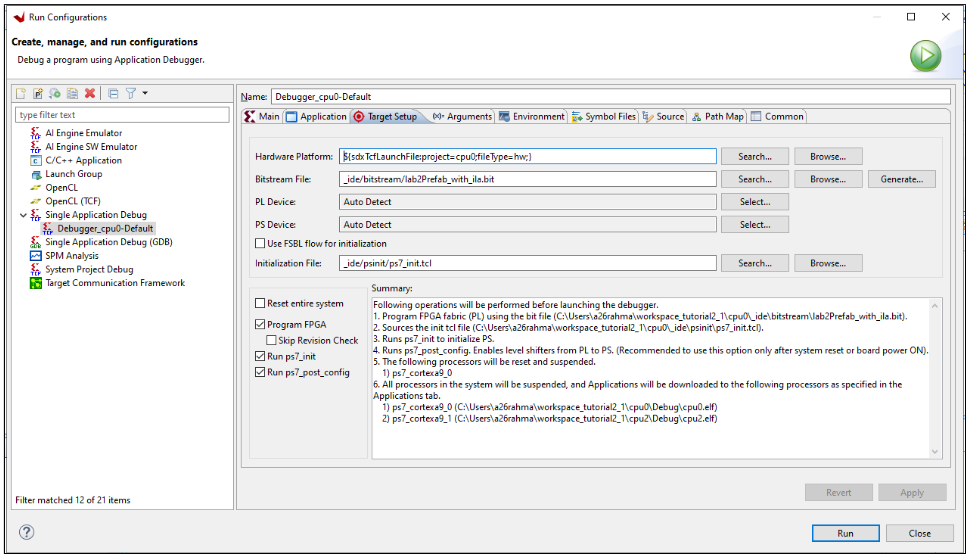
You also need to add: `Xil_SetTlbAttributes(<Your shared address start>,DEVICE_MEMORY);` at top inside your main function. This sets the necessary TLB attributes in MMU to allow spinlocks to work in 4KB of region starting with start address.

Regarding the circular buffers for vdma, you have three options: (1) you can manually allocate the full circular buffer in the shared area; (2) you can malloc individual data structures on one of the cores, and then put the corresponding pointers in the circular buffer, which can again be allocated in the shared area; (3) you can malloc the circular buffer on one of the cores, and then pass a pointer to its address to the other core through a variable allocated in the shared area.

6.2.4 Execute the Software Application

To execute the dual core application, you need to first build the application on both cores, in either Debug or Run mode. You can use the Run Configuration from the previous Assignments or you can follow the same steps to create a new Run Configuration. Follow these steps to specify the loader to put second application on the second core.

1. In the Target tab unckeck the box for 'Reset entire system', make sure other settings are correct and correct bitfile is selected.



Name: Debugger_cpu0-Default

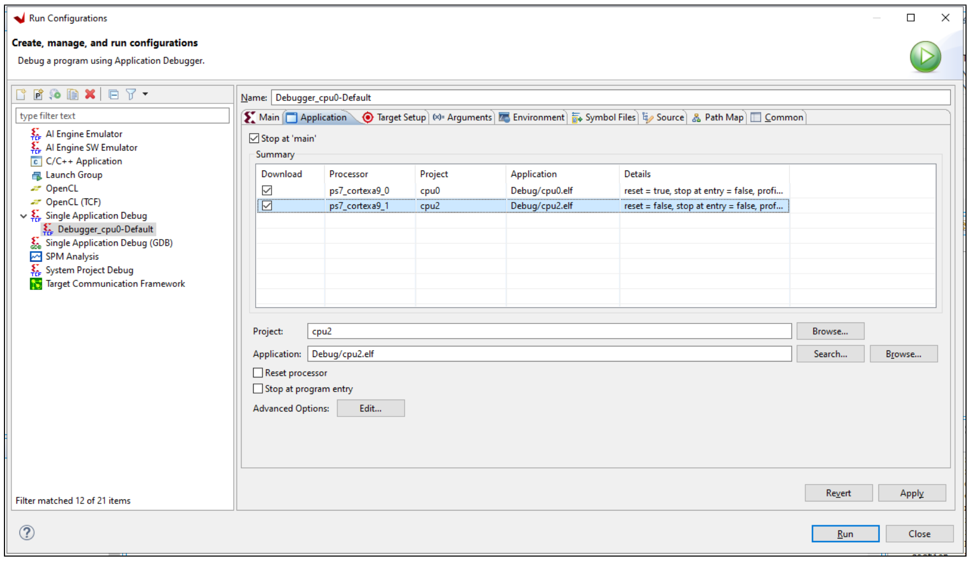
Hardware Platform: ${sdxTcfLaunchFile:project=cpu0;fileType=hw;}
Bitstream File: _ide/bitstream/lab2Prefab_with_ila.bit
PL Device: Auto Detect
PS Device: Auto Detect
☐ Use FSBL flow for initialization
Initialization File: _ide/psinit/ps7_init.tcl

☐ Reset entire system
☑ Program FPGA
☐ Skip Revision Check
☑ Run ps7_init
☑ Run ps7_post_config

Summary:
```
Following operations will be performed before launching the debugger.
1. Program FPGA fabric (PL) using the bit file (C:\Users\a26rahma\workspace_tutorial2_1\cpu0\_ide\bitstream\lab2Prefab_with_ila.bit).
2. Sources the init tcl file (C:\Users\a26rahma\workspace_tutorial2_1\cpu0\_ide\psinit\ps7_init.tcl).
3. Runs ps7_init to initialize PS.
4. Runs ps7_post_config. Enables level shifters from PL to PS. (Recommended to use this option only after system reset or board power ON).
5. The following processors will be reset and suspended.
   1) ps7_cortexa9_0
6. All processors in the system will be suspended, and Applications will be downloaded to the following processors as specified in the
Applications tab.
   1) ps7_cortexa9_0 (C:\Users\a26rahma\workspace_tutorial2_1\cpu0\Debug\cpu0.elf)
   2) ps7_cortexa9_1 (C:\Users\a26rahma\workspace_tutorial2_1\cpu2\Debug\cpu2.elf)
```

Figure 6.5: Building the project

2. In the Application tab in the Run Configuration, check the boxes for both the cores and make sure that correct project and corresponding elf files are displayed for both the cores.


Run Configurations

Create, manage, and run configurations

Debug a program using Application Debugger.

Name: Debugger_cpu0-Default

☑ Stop at 'main'

Summary

Download	Processor	Project	Application	Details
☑	ps7_cortexa9_0	cpu0	Debug/cpu0.elf	reset = true, stop at entry = false, profi...
☑	ps7_cortexa9_1	cpu2	Debug/cpu2.elf	reset = false, stop at entry = false, prof...

Project: cpu2

Application: Debug/cpu2.elf

☐ Reset processor

☐ Stop at program entry

Advanced Options: Edit...

Figure 6.6: Building the project

3. Make sure that 'Reset processor' checkbox is unchecked for second core (ps7_cortexa9_1) and is checked for first core(ps7_cortexa9_0).

Now you should be able to run the application on both the cores. For the dual core application, it is recommended to halt the 'slave' core while the 'master' core performs the initial configuration such as mallocing and initialization of devices. Then the other core can be started by unlocking the spinlock.

6.3 Task 2: Implement the parallel MJPEG423 decoder

Following the schedule derived in Task 1, parallelize the MJPEG423 application and run it on the Zynq board. As in Assignment 1-2, you need to support playback control and display the frames periodically using the timer handler; make sure to check the result of vdma_out to ensure that there are no buffer underflows. Your goal is to meet 24 fps, or alternatively come as close to possible to it.

Note 1: the schedule derived in Task 1 is based on a model of the application. Assuming that both the profiling and the schedule design was performed correctly, the implemented application should meet the targeted frame rate. However, implementing an application on a heterogeneous multicore system is always tricky, as execution times can be affected due to various overheads and resource contention. For this reason, we suggest you to keep the profiling code you developed for Assignment 1 (preferably by using ifdefs to allow for conditional compilation) so that it can be used to check the application timing if problems arise.

Note 2: to simplify the implementation of the playback control, do not code a different schedule for the first/last frame you display compared to intermediate frames. Instead, simply skip executing a task if it belongs to a frame before/after the sequence you want to display (e.g., if your schedule performs the color conversion of frame $x-1$ after the SD Read of I-frame x, then after fast-forwarding to frame x and executing its SD Read, your logic can see that frame $x-1$ is not part of the sequence and simply skip running the CC, while still locking/unlocking relevant spinlocks to maintain schedule synchronization).

6.4 Deliverables

Demo: during the demo, you need to demonstrate the ability to run the parallel application at the targeted, constant frame rate, including playback control.

Report: write a short report providing: (1) the resulting DAG of precedence constraints (draw at least one I-frame and the following P-frame). Make sure to clarify which computation is performed in each task. (2) A drawing of your schedule (specifying the obtained periodicity). (3) Compute the worst-case makespan of any frame, the required schedule delay, and the total buffer space. Clearly show your computations.

The mark breakdown is reported in Table 6.1. For the demo component, you will be marked based on correctness. For the optimization component, you will be marked on a scale based on your ability to come close or meet 24fps. **Note:** reaching 24fps is challenging. You can still get a high mark even if you come close, but do not match the intended frame rate. Do not spend an unbounded amount of time trying to optimize the last fps. Similarly, unless there are insurmountable issues with the schedule you developed in Task 1, we do not suggest doing large modifications. If you find that your Task 1 schedule is problematic and changes are required, please speak to the teaching team first.

Total Assignment 3	9%
DAG, Schedule, Makespan	3%
Demo	4%
Optimization	2%

Table 6.1: Mark Breakdown For Assignment 3

Appendix A

ECE 423 API Reference

This appendix provides the API description for SD card, video display, timer, push buttons and cache management.

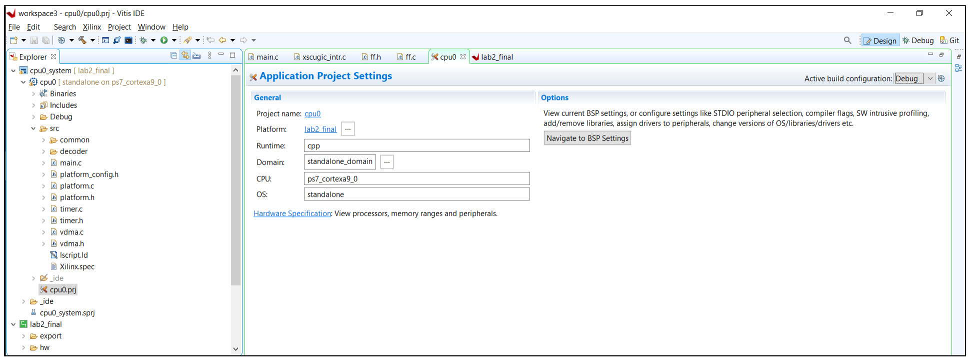
All the functions described in the following sections follow this convention for their arguments and return values that are pointers.

A.1 SD Card API

Configuring BSP Settings

In order to enable the library for SD Card, we need to modify Board Support Package (BSP) settings.

Open the .prj file and click "Navigate to BSP settings". Then, "Modify BSP Seettings" and check the box for xilffs library.



Application Project Settings

Active build configuration: Debug

General

Project name: cpu0

Platform: lab2_final

Runtime: cpp

Domain: standalone_domain

CPU: ps7_cortexa9_0

OS: standalone

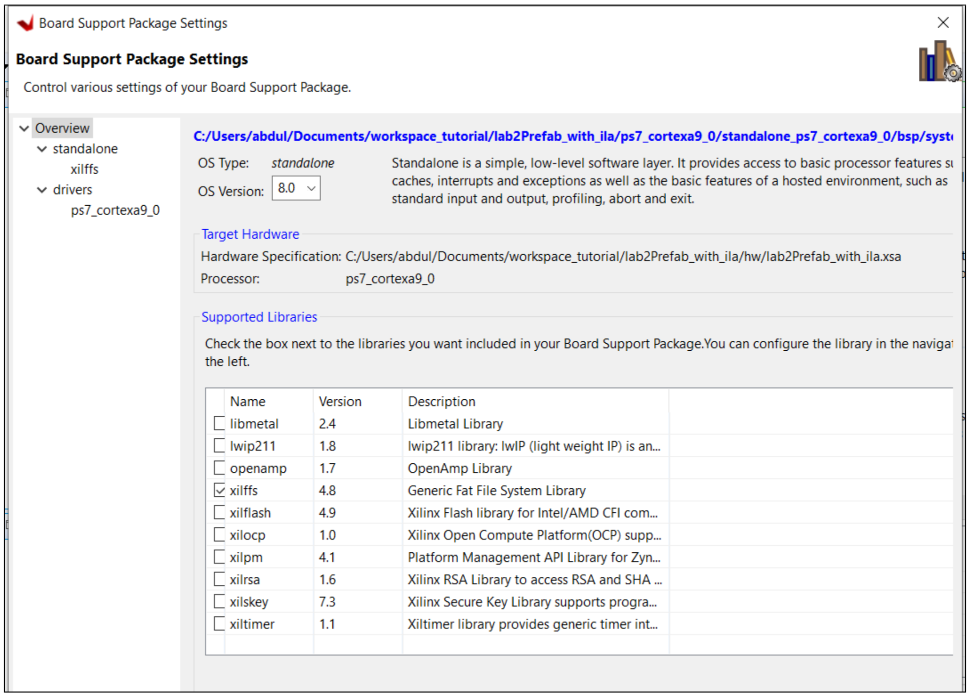
Hardware Specification: View processors, memory ranges and peripherals.

Options

View current BSP settings, or configure settings like STDIO peripheral selection, compiler flags, SW intrusive profiling, add/remove libraries, assign drivers to peripherals, change versions of OS/libraries/drivers etc.

Navigate to BSP Settings

Figure A.1: Navigating to BSP settings (1)



Board Support Package Settings

Board Support Package Settings

Control various settings of your Board Support Package.

- Overview
  - standalone
    - xilffs
  - drivers
    - ps7_cortexa9_0

C:/Users/abdul/Documents/workspace_tutorial/lab2Prefab_with_ila/ps7_cortexa9_0/standalone_ps7_cortexa9_0/bsp/syst

OS Type: *standalone*

OS Version: 8.0

Standalone is a simple, low-level software layer. It provides access to basic processor features su caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.

Target Hardware

Hardware Specification: C:/Users/abdul/Documents/workspace_tutorial/lab2Prefab_with_ila/hw/lab2Prefab_with_ila.xsa

Processor: ps7_cortexa9_0

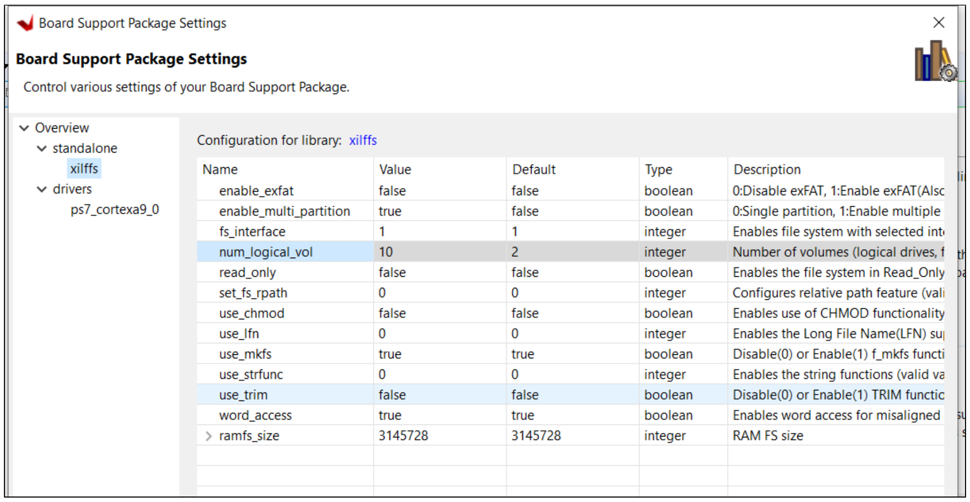
Supported Libraries

Check the box next to the libraries you want included in your Board Support Package.You can configure the library in the navigat the left.

Name	Version	Description
☐ libmetal	2.4	Libmetal Library
☐ lwip211	1.8	lwip211 library: lwIP (light weight IP) is an...
☐ openamp	1.7	OpenAmp Library
✓ xilffs	4.8	Generic Fat File System Library
☐ xilflash	4.9	Xilinx Flash library for Intel/AMD CFI com...
☐ xilocp	1.0	Xilinx Open Compute Platform(OCP) supp...
☐ xilpm	4.1	Platform Management API Library for Zyn...
☐ xilrsa	1.6	Xilinx RSA Library to access RSA and SHA ...
☐ xilskey	7.3	Xilinx Secure Key Library supports progra...
☐ xiltimer	1.1	Xiltimer library provides generic timer int...

Figure A.2: Navigating to BSP settings (2)

In left panel, open xilffs and set the "enable_multi_partition" field to "true" and "num_-logical_vol" to 10 as shown in figure A.3. After this, you should be able to include the FatFS library.



Board Support Package Settings

Board Support Package Settings

Control various settings of your Board Support Package.

- Overview
  - standalone
    - xilffs
  - drivers
    - ps7_cortexa9_0

Configuration for library: xilffs

Name	Value	Default	Type	Description
enable_exfat	false	false	boolean	0:Disable exFAT, 1:Enable exFAT(Also
enable_multi_partition	true	false	boolean	0:Single partition, 1:Enable multiple
fs_interface	1	1	integer	Enables file system with selected inte
num_logical_vol	10	2	integer	Number of volumes (logical drives, f
read_only	false	false	boolean	Enables the file system in Read_Only
set_fs_rpath	0	0	integer	Configures relative path feature (vali
use_chmod	false	false	boolean	Enables use of CHMOD functionality
use_lfn	0	0	integer	Enables the Long File Name(LFN) sup
use_mkfs	true	true	boolean	Disable(0) or Enable(1) f_mkfs functi
use_strfunc	0	0	integer	Enables the string functions (valid va
use_trim	false	false	boolean	Disable(0) or Enable(1) TRIM functic
word_access	true	true	boolean	Enables word access for misaligned
> ramfs_size	3145728	3145728	integer	RAM FS size

Figure A.3: Configuring xilffs library

Library Introduction

The Xilinx SDK integrates the generic FatFs library available at `http://elm-chan.org/fsw/ff/00index_e.html`. FatFs is a lightweight software library for microcontrollers and embedded systems that implements FAT/exFAT file system support. The library is well documented and example usage is provided along with the functions.

Include File

`#include "ff.h"`

Functions to be used for lab

`f_mount`	used to mount the SD card parition
`f_readdir`	used to read items in directory sequentially
`f_open`	used to open a file
`f_read`	reading data from the file
`f_lseek`	seeking into the file
`f_close`	closing the file

Important note on partitions

The SD card has 3 partitions out of which ECE423 only uses the 3rd partition. To make sure the system mounts and accesses the correct parition, the parition is to be passed in both the functions for mount and the function for opening the file. An example usage of FatFs for ECE423 is shown below:

```
#include "ff.h"
static FATFS fatfs;
static FIL fil;

void main(){
    // mount the correct partition
    f_mount(&fatfs, "3:/", 1);

    // open the mpg file
    f_open(&fil, "3:/v1_1730.mpg", FA_READ);
```

```
        // read the number of frames from mpg file
        uint32_t num_frames;
        uint32_t NumBytesRead;
        f_read(&fil, (void*)&num_frames, sizeof(uint32_t), (UINT *)&NumBytesRead);
    }

    // NOTE: return value of f_mount, f_open and f_read functions may be used to determine the status
        of execution of these operations.
```

Sample function for listing files

Note: the way this function is built, it function might not output the exact names. This is for reference only.

```
    FRESULT scan_files ()
    {
        FRESULT res;
        DIR dir;
        UINT i;
        static FILINFO fno;


        res = f_opendir(&dir, "3:/");                        /* Open the directory */
        if (res == FR_OK) {
            for (;;) {
                res = f_readdir(&dir, &fno);                   /* Read a directory item */
                if (res != FR_OK || fno.fname[0] == 0) break;  /* Break on error or end of dir */
                if (!(fno.fattrib & AM_DIR)) {                    /* It is a directory */
                    printf("%s/%s\n", "3:/", fno.fname);
                }
            }
             f_closedir(&dir);
        }

        return res;
    }
}
```

A.2 Video Display API

Include File

`#include "ece423_vid_ctl/ece423_vid_ctl.h"`

Functions

`vdma_init()`		
Prototype:	`uint32_t vdma_init(uint32_t width, uint32_t height, uint32_t frame_buff_size)`	
Description:	Initialize the video display. Allocate memory for frame buffers.	
Arguments:	`width`	Video width.
	`height`	Video height.
	`frame_buff_size`	Number of frame buffers.
Return:	1 when runs successfully. 0 when error occurs in allocating frame buffers.	

`buff_next()`		
Prototype:	`rgb_pixel_t* buff_next()`	
Description:	Get the next buffer to process data into.	
Arguments:	`void`	
Return:	Pointer to buffer of type rgb_pixel_t* with width and height defined by arguments passed in vdma_init(). NULL in case of buffer overflow.	

`buff_reg()`		
Prototype:	`uint32_t* buff_reg()`	
Description:	Mark the buffer ready to be displayed.	
Arguments:	`void`	
Return:	1 when the buffer is successfully marked as completed. 0 In case of underflow.	

`vdma_out()`		
Prototype:	`uint32_t vdma_out()`	
Description:	Display next frame available in buffer.	
Arguments:	void	
Return:	1 if new frame was was popped out and displayed. 0 if frame buffer underflows.	

`vdma_close()`		
Prototype:	`void vdma_close()`	
Description:	Free the frame buffer memory.	
Arguments:	void	
Return:		

Typical Implementation

```
#include " ece423_vid_ctl / ece423_vid_ctl .h"
void main(){
    vdma_init(1280, 720, 5);
     rgb_pixel_t * rgbblock;
    //loop over all the frames
    rgbblock = buff_next();
    //read new frame from SD card and process it
    //call buff_reg () to mark the frame as completed
     buff_reg ();
    //display
    vdma_out();
    //end loop
    //free memory at the end of program
    vdma_close();
}
```

A.3 Timer and Gpio API

Include File

`#include "timer_gpio.h"`

Functions

timer_gpio_init()		
Prototype:	`int timer_gpio_init(void (*isr_timer)(void* data), void (*isr_gpio)(void* data));`	
Description:	Initialize the timer and gpio with TimerISR and GpioISR.	
Arguments:	`void (*isr_timer)(void* data), void (*isr_gpio)(void* data)`	Pointer to TimerISR and GpioISR. See listing below for example usage.
Return:	Returns 1 when initialization fails, 0 when succeeds.	

timer_start()		
Prototype:	`void timer_start(unsigned int count)`	
Description:	Start the timer with the given count. The count value to absolute time is determined by XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ divided by 2 parameter available in xparameters.h. If XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ is 650000000, passing a value of (650000000)/(2*1) would allow for timer to trigger every second. (650000000)/(2*n) would generate interrupt n times every second	
Arguments:	`count`	Count value
Return:		

`timer_stop()`		
Prototype:	`void timer_stop()`	
Description:	Stop the timer.	
Arguments:	None	
Return:		

`read_pin()`		
Prototype:	`int read_pin();`	
Description:	Read the push buttons (BTN0 - BTN3) available on PYNQ board.	
Arguments:	None	
Return:	Returns an integer based on the push button that is pressed. If no button press is detected, return -1. e.g: Return value will be 0 when BTN0 is pressed. See example listing for usage.	

Typical Implementation

```
#include "xparameters.h"
#include " xil_types .h"
#include "timer_gpio.h"

#define TIMER_1S  325000000 //1 second
#define TIMER_FPS  1

 volatile  int8_t  val_0;
 volatile  int8_t  val_1;

void GpioHandler(void *CallBackRef, u32 Bank, u32 Status)
{
   val_0  = read_pin();
}

void TimerHandler(void*){
   val_1  = 1;
}
int main()
{
   val_0  = -1;
```

```
    val_1  = 0;

    // initialize  timer and start Gpio with  interrupts
      timer_gpio_init (TimerHandler,GpioHandler);
     //start  timer
     timer_start (TIMER_1S/TIMER_FPS);

     while  (1){
       if ( val_0  != -1){
         printf ("Button Pressed: %d\n\r",val_0);
         val_0  =-1;
       }
       if ( val_1  == 1){
         val_1  = 0;
         print ("Timer Triggered!  \n\r");
       }
     }
     return  0;
}
```

A.4 Cache API

Include Files

`#include "xil_cache.h"` `#include "xil_cache_l.h"`

Library Documentation

The Xilinx SDK provides API for access to cache related operations of Cortex A9 processors. Documentation and function prototypes for both the libraries are given at `https://docs.xilinx.com/r/en-US/oslib_rm/Arm-Cortex-A9-Processor-Cache-Functions`. A list of useful functions that may be used for the lab are provided below. It is not necessary to use all the functions.

Functions to be used for the lab

`Xil_L1DCacheFlush`	Flush L1 Data Cache
`Xil_L1DCacheInvalidate`	Invalidate L1 Data Cache
`Xil_L2CacheFlush`	Flush L2 Cache
`Xil_L2CacheInvalidate`	Invalidate L2 Cache
`Xil_L1DCacheFlushRange`	Flush a range of addresses in L1 Data Cache
`Xil_L1DCacheInvalidateRange`	Invalidate a range of addresses in L1 Data Cache
`Xil_L2CacheFlushRange`	Flush a range of addresses in L2 Cache
`Xil_L2CacheInvalidateRange`	Invalidate a range of addresses in L2 Cache