Design of an Asynchronous Ditherer for an MPEG Decoder

Yanyi Zhao
Electrical and Computer Engineering Department
University of Utah
Salt Lake City, UT 84112

April 14, 2003

Abstract

The goal of this project is to design an asynchronous ditherer for an MPEG decoder. The design begins with a software description of the ditherer algorithm found in a software MPEG decoder [2]. Then, this description is migrated from a software implementation to a mixed hardware/software design, where the hardware can be a FPGA or an IC circuit design. The MPEG algorithm is initially described using C/C++ code. This C/C++ code is partitioned into a hardware ditherer and a software component described using VHDL (a hardware description language). The style of VHDL includes the use of a channel package to help model the asynchronous communication between the hardware and software. At this point, CAD tools for asynchronous and physical design are leveraged to synthesize a hardware implementation of the ditherer, and to implement the communication between hardware and software components. We plan to use this design as a case study for the development of design methods for hardware/software co-design of asynchronous systems.

1 Introduction

The goal of this project is to design an asynchronous ditherer for an MPEG decoder [1]. MPEG stands for Moving Picture Expert Group, which is a group of people that meet under the International

```
Open MPEG stream
For each PictureGroup in stream do
    For each Picture in PictureGroup do
        For each Slice in Picture do
            Until end of Slice do
                Read and process a Macroblock of data
                Processing fills in Frame arrays with new data
            " Dither " frame
        Display dithered frame
Close MPEG stream
```

Figure 1: MPEG algorithm.

Standards Organization (ISO) to generate standards for digital video and audio compression. As the famous video compression standard, MPEG is used in many different current and emerging products, such as HDTV, DVD players, video conferencing, internet video, and many other applications. The reason that MPEG is widely used is because compared to other video compression standards, MPEG files require less storage space for the same quality. This is because MPEG uses sophisticated compression techniques. So far, MPEG has several generations, including MPEG-1, MPEG-2, and MPEG-4. MPEG was finalized in 1991, and it is the basic standard algorithm for the other generation, both MPEG-2 and MPEG-4 are based on it [5] .

There are two important parts in the MPEG-1 algorithm, the encoder and decoder. The decoding progress for an MPEG picture stream is really a classic "top down functional decomposition problem". What we mean can be seen in Figure 1.

Each picture group is made of many pictures, each picture is made of many slices, each slice is made of many macroblocks, and each macroblock is made of many frame arrays. At the end of the slice, it reads and processes a Macroblock of data, and use new data to fill into the frame arrays. Then it dithers the frames, and display all the dithered frames onto the screen. At the end, the MPEG stream is closed. This MPEG-1 decoder is described using C++ code [3].

2 Partitioning

The complete design flow of this project is shown in Figure 2 as the following, the design begins with a software description of the ditherer algorithm found in a software MPEG decoder. Then this description is migrated from a software implementation to a mixed hardware/software design, so we can improve the performance of the algorithm. Therefore, after selecting an algorithm, our method analyzes the algorithm to determine which part should be done in software and which part should be done in hardware, which is the partitioning step. Next, the software component is implemented using C++ code, and the hardware component implemented using VHDL code and synthesized to a FPGA or ASIC. After we get these two parts done, we integrate them together, in order to make them communicate with each other. Our project follows this design flow but focuses on the hardware parts.

There are many classes for this decoder program, such as bitstream.cc, block.cc, frame.cc, MPEGdecoder.cc, windows.cc, ditherer.cc, and so on [4]. The program spends its time as follows: the largest fraction of the time is spent in the ditherer, which is at least 15 percent. Around 8-10 percent of the time is spent in the frame function. The IDCT code (JPEG's implementation) is about 5 percent. The write function is around 5 percent; read may be about 1 percent most other functions are less than 1 percent. Since the ditherer uses the most time, we chose that function for hardware implementation. Dither is a graphic word, which means to make the pictures smooth and not too blocky. The ordered2 dither algorithm uses a complex data structure for mapping luminance-chrominance triples into pixel numbers. This complex data structure is built in the constructor, and released in the destructor. Other ditherer subclasses could be defined to implement some of the alternative dithering processes provided with the MPEG program. The ordered2 dither is the default; it is faster than most of the alternatives. So we can see that the ditherer is the most complicated part during the decoding process. In order to see what the ditherer function really does in an MPEG-1 decoder algorithm, we played several MPEG movies on it, which helps us to understand how this ditherer function works. The input

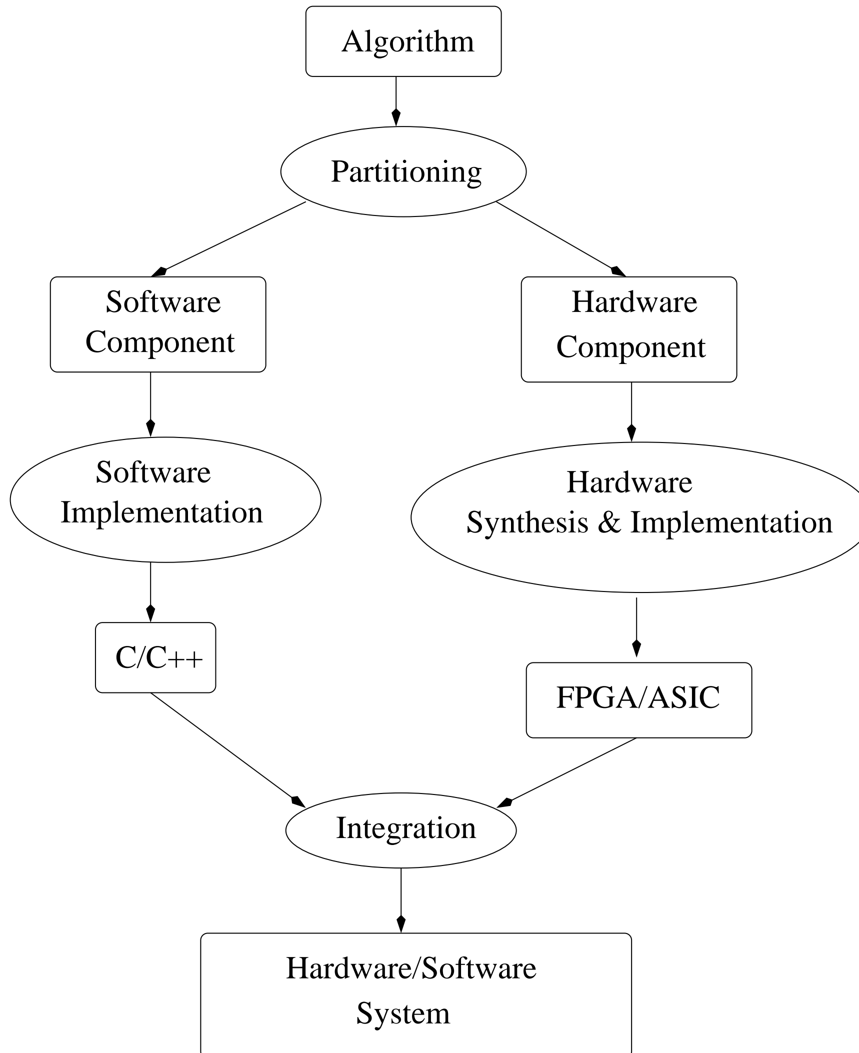

Figure 2: Design flow.

of the ditherer function is a frame that is described using luminance (Lum), and chrominance (C_b, C_r) data. It uses a translation table to map Lum, C_r, C_b into 8−bit RGB color maps. The ditherer function also smooths the image in order to make the picture not too blocky.

3 Software Implementation

First, we begin with a software implementation for the MPEG decoder. We want to isolate the ditherer function into its own software implementation. However, if we do that, the ditherer function

cannot be tested using MPEG files. Therefore, we need to create input and output data for the ditherer function for several random frames for later testing on the software and hardware implementation.

In order to create data files for testing this ditherer function independent of the rest of the decoder, we first played a movie named "Canyon "using the algorithm, and then found out there is a total of 1056 frames in the movie. We also output the height, width, and page size of each frame, which would be useful for later analysis. Second, we randomly selected ten example frames out of the 1056 and stored the input and output of the dither function to several files. We also stored the translation table to a file. If this data is stored naively, it would require a huge amount of memory 16*256*256*256 bytes or 2^{28} bytes, which is 256M bytes. But in reality, we do not really need as large a memory as this since many entries are duplicated. The reason is explained below. We re-calculated the data, we determined the actual size of each translation table to be 256*4*4. Since we have 16 ditherer arrays, each ditherer array points to one translation table, so the total memory size is only 256*4*4*16, which is 64K bytes.

Because some design and optimization ideas are very different between a software and hardware implementation, we must redesign the algorithm for hardware based on what we understand the original MEPG-1 decoder algorithm from C++ source code is doing. Otherwise, it is very difficult to translate C++ code to VHDL code directly. The two key elements that must be translated to hardware are the *TranslationTable* and *ProcessImage* function. The C++ code is really difficult to understand. Pointers are not possible in hardware, so we must re-design these two elements in order to better map to a hardware implementation. The functions must also be rewritten so that ditherer.cc can be compiled and simulated alone.

The MPEG-1 algorithm uses YUV color space (Y C_r C_b) to represent the images. If an image is stored in another form, it must be converted to YUV format first. All the images are represented as 24 bits per pixel that are divided into 8 bits for luminance information (Y), 8 bits for each part of

the Chrominance information (C_r C_b). When it converts from the other formats to YUV format, the luminance (Y) stays the same, but the C_r C_b have to be reduced by a ratio of 4:1. This sampling does not affect the quality of the image, because human eyes are more sensitive to luminance than chrominance. So basically, the TranslationTable translates all the input data Lum, C_r, C_b, DA into 8-bit RGB value. Each frame has 16 DA (Ditherer Array), each of the DA points to a cube with Lum, C_r, C_b as its three dimensions. Each of them is an array of size 256. By looking at the input data files, we realized that C_r and C_b are not pointing to 256 different addresses, instead, they are separated into four different sections. Let's use the first DA, DA_0, as an example, since it is the basic DA among the 16 DAs, all of the rest of the 15 DAs are shifted by 4 entries from this. The structure of DA_0 is shown in Figure 3. The ditherer function receives 8-bit values for C_b, 8-bit values for C_r, and 8-bit values for Lum, and uses these values to index into a three-dimensional dither array. Although the 8-bit C_b value maps to 256 entries, the C_b value is actually only used to map into 4 different sections. By looking at Figure 3, any number between 0 to 32 is aliased to the first C_b section. Each section of C_b points to the same dashed box. Similarly, the entries from 33 to 96 also aliased to the same section, and point to a unique dashed table. Similarly, any entries from 97 to 160 are aliased to the same section, and any entries from 161 to 255 are also aliased to the same section. So in reality, even though there are 8 bits of resolution for C_b, it is mapped into 4 sections so it has an effective resolution of only 2 bits. Similarly for C_r, even though there are 8 bits of value for C_r, mapping into 256 entries, but it is separated into 4 different sections. If we assume C_b as the X-axis, C_r as the Y-axis, Lum as the Z-axis, the cutoff points for C_r is the same with C_b — 33, 97, 161. So any entry between 0 to 32 are aliased to the same section, and map to the same luminance table. Any entry between 33 to 96 are aliased to the same luminance table, any entry between 97 to 160 are aliased to the same luminance table, and any entry between 161 to 255 are aliased to the same luminance table. So once again, the original 8 bits of resolution for C_r is mapped into 4 different sections, so it has an effective resolution of only 2 bits. For

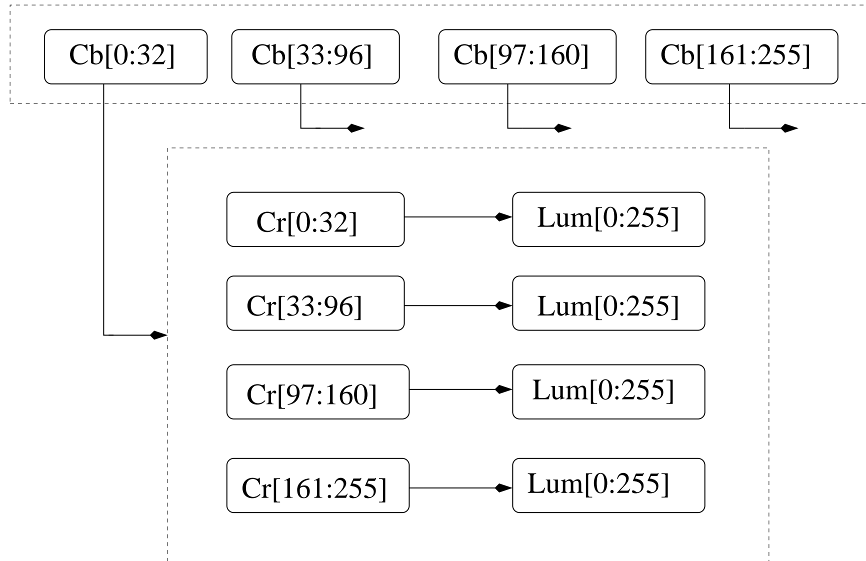

Figure 3: The structure of DA_0.

luminance, however, the full 8 bits of resolution is used. Therefore, if we add all the resolutions of C_b, C_r, and *lum* together, we only need ($2^2 * 2^2 * 2^8 = 12$ bits). Since we have 16 ditherer arrays, we have 16 bits of resolution total which requires only 64K bytes of memory. Compare this number with what we had before, 256M bytes, it is much smaller. This result also perfectly matches the assumption of the original design that the spatial resolution of C_b:C_r:Lum = 1:1:4.

Since we know the exact structure of the *TranslationTable*, we can write our own code that does the same thing without pointers. This way, we do not need any other files in order to run the *TranslationTable*, also the structure is much more clear and easier to understand. To test whether we did it right, we compare the output data we get from the new structured ditherer program with the output data from the original ditherer program. They are exactly the same, which means we have been successful on implementing the *TranslationTable*.

Next, we re-designed the *ProcessImage* function. This function has three pointers and three big for loops. The original function is really confusing because of the order that the author uses to map the 16 *DA*s. There are three for-loops in the original design, the outside one is to jump every four rows of data, the inside two for-loops are for all of the 16 *DA*s, with one for even number *DA*s, and

one for odd number *DA*s. So, the order of *DA* used is 0,8,12,4,2,10,14,6,3,11,15,7,1,9,13,5. Since we need to implement it into hardware design, we have to restructure the function without any for loops and pointers. After we have done that, we are going to compare the result with the original output data to check the correctness. Consider an example frame that is 4 pixels by 4 pixels. The structure is shown in Figure 4. All of the black letters stand for the values of C_b, C_r, and lum for each pixel. The numbers in the upper left hand corner of the lum table stand for which dither array we are going to use. The *DA*s are ordered in a special way, which allows for a simple implementation using 2 XOR gates. Since the spatial resolution between Lum:C_b:C_r is 4:1:1, we can see from the figure here, each C_b, C_r value corresponds to 4 luminance values. First the *ProcessImage* function takes a C_b value A, C_r value E, and *Lum* value I, then indexs them into dither array 0 from the translation table, to find out the 8-bit RGB value that corresponds to these luminance and chrominance values. Next, we reuse the C_b and C_r value, but move on to the new *Lum* value J, index them into dither array 8, to find the 8-bit RGB value that corresponds to these luminance and chrominance values. Then, we again reuse the em C_b and C_r value, but move on to the new *Lum* value M, index them into dither array 12, to find the 8-bit RGB value. Finally, we use the Lum value O, but the same C_b and C_r value, and index them into dither array 4 to find the 8-bit RGB value. After the first C_b,C_r value have been completely used with 4 luminance values, we obtain a new C_b value B and a new C_r value F, and repeat the same process for the *Lum*. The whole process keeps repeating until all the luminance and chrominance values have been processed.

After we know how this *ProcessImage* function works, we re-write the function using our own technique, which is without any for loops and pointers. This way makes it much easier for the implementation in hardware, also it is easier to understand the structure of the function. To test the correctness of our re-designed function, we input the data into our ditherer program, and then compared this new output data with the original output data that we get for the movie "Canyon".

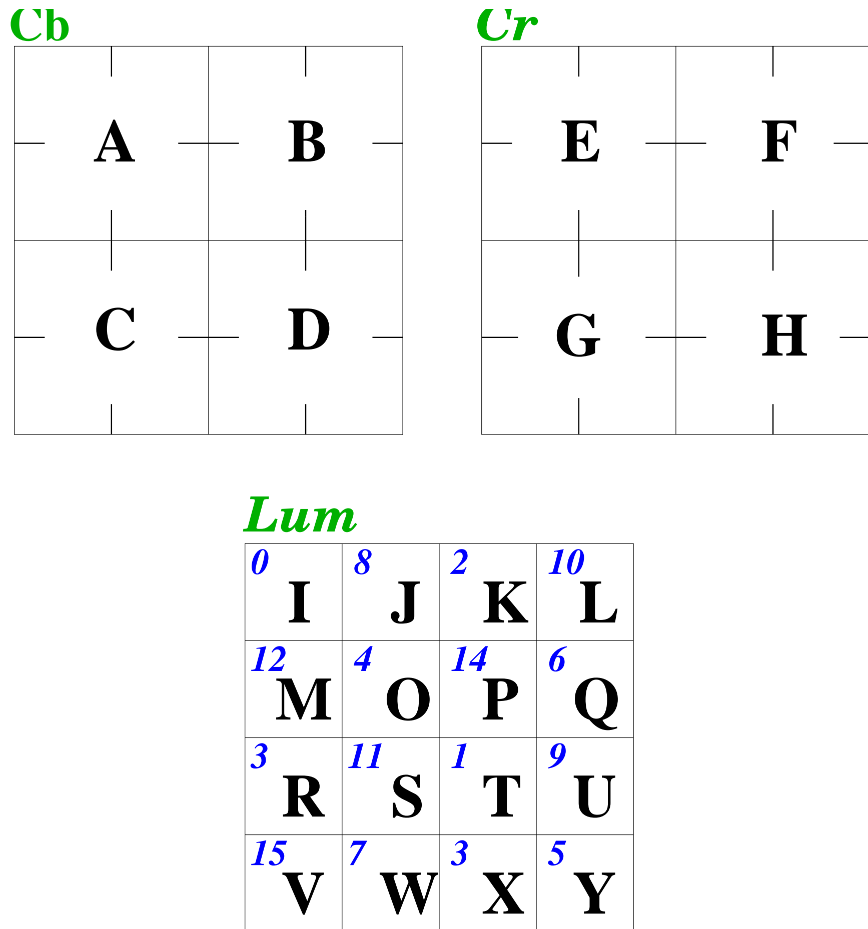

Figure 4: Process image structure.

The results are exactly the same.

Besides the above two functions, we have also made some small changes to some other functions in this ditherer program in order to make the whole program consistent. After extracting the ditherer function, we simulate it by comparing the original output and the output we got after re-designing the algorithm. We found that the output frames generated from our new ditherer to exactly match the original output frames.

4 Hardware Behavioral Implementation and Simulation

After we restructure the software implementation for the ditherer function, we translated the C++ code into behavioral VHDL code for modified ditherer function. We started to break down our re-designed algorithm into hardware components by using VHDL code (a hardware description language). The style of the VHDL includes the use of a channel package to help model the asynchronous communication between the hardware and software. We have thought about a few designs, and the best one we have now is taking this re-designed ditherer file into a control circuit and the datapath. For the control circuit, we used channel level VHDL code, which controls the datapath to receive data, calculate the data, and send data, etc. For the datapath part, we broke down the ditherer program into several manageable sub-components, such as a latch for Lum, a latch for C_b, a latch for C_r, $DACalculation$, $TranslationTable$, Memory, and Counter. Most components are also asynchronous designs using channel level VHDL code. All of these have been simulated by using ModelSim. In order to simulate this hardware implementation, we need to have a test bench for the whole design, which basically sends the *lum.dat*, *cb.dat*, *cr.dat*, *DA.dat* to the datapath, and after all the calculations in the hardware implementation, the hardware sends the output data to the test bench to compare with the result we have from the software implementation. After the simulation, all of the results matched successfully.

After this channel-level VHDL, we go even deeper to handshaking level VHDL code to implement this hardware design. We need to reveal all of those required handshaking signals that are hidden by ATACS [6]'s channel package. After completing the handshaking level VHDL, we also need to check the correctness of the implementation, which can be done the same way as we did for the channel-level VHDL. We design a test bench for the whole hardware design, and then use ModelSim to simulate it. Since we have already done the C++ implementation and channel level VHDL simulation, we can simulate this handshaking-level design to against those implementations and simulation results. This

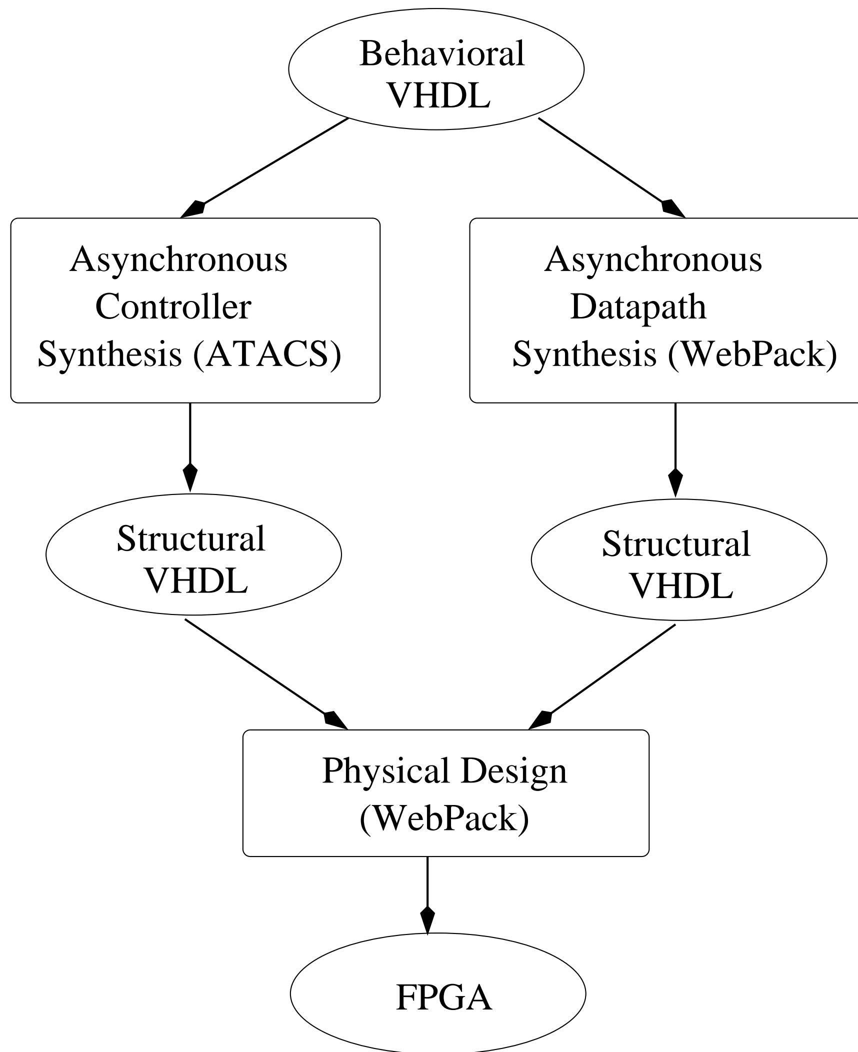

Figure 5: Hardware synthesis and implementation.

part has also been tested successfully.

5 Hardware Synthesis and Implementation

Until now, we basically have had a correctly implemented and simulated design for the ditherer program except we have not synthesized it yet. Next, the hardware synthesis and implementation steps can be seen in Figure 5. We take the behavioral VHDL code, and partition them into an asynchronous controller and datapath. For the controller, we synthesize it using ATACS which is an asynchronous synthesis tool developed in our research group. The controller can be synthesized by

using the ATACS tool, which generates ATACS standard-logic-based synthesized equations, which can be translated into a control circuit built of gC elements and standard logic gates. For the datapath, we synthesized it using Xilinx's Webpack software. After we synthesized it, we obtain structural VHDL code which we combine together, and then physically design on to a FPGA by using Webpack. We took the handshaking-level VHDL, and translated it into synthesizable VHDL code, and then created schematics for all of the components in the datapath, then implemented it into an FPGA. Most of the time, the handshaking-level VHDL code required some modification to meet the synthesizable VHDL specification. We have realized that the simulation VHDL and synthesis VHDL are not the same codes; thus, later modification to either one may cause inconsistency in the design. With all these considerations, we did translate all of the handshaking-level VHDL codes in the datapath into synthesizable VHDL code, and we got all of the components we need in order to build the datapath. The details for the datapath is shown in Figure 6, and the complete hardware design is shown in Figure 7 as the following. For the datapath, after reset, the counter is initialized to 0, the memory is set to write mode. Next, the software provides the first entry of the translation on the Din wire for the memory, then MEM_REQ is set high to write this entry into the translation table and store it into memory. There is a delay element, which is connected between MEM_REQ and MEM_ACK signals. The size of this delay element is set to match the time that it takes to write an entry into memory. In other words, after MEM_REQ goes high, MEM_ACK becomes high only after a sufficient amount time has passed such that we are certain that the entry has been successfully written into the memory. Next, CNT_REQ is set high in order to increment the address to point to the next entry in the memory. After CNT_ACK goes high, the software provides the next entry for the translation table and the control sets MEM_REQ high to write it into the memory. This process repeats until the memory is filled with the whole translation table. At this point, we need to move to the other side of the datapath. After reset, the DACAL (ditherer array calculation) module is set to point to

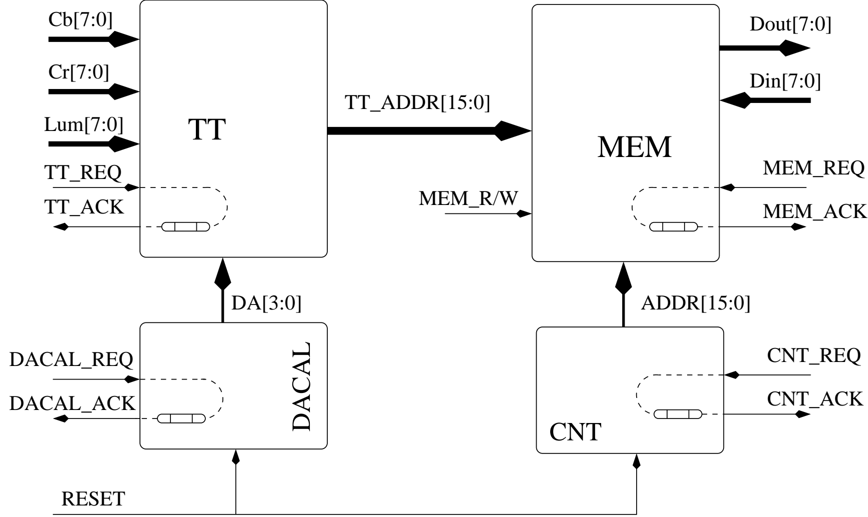

Figure 6: Datapath.

DA_0. Next, the software sends the 8-bit value for each C_b, C_r, and *lum* values, when all the data is ready, TT_REQ is set to high. The TT module then uses the 28-bits provided by C_b, C_r, *Lum*, and DA to determine the correct 16-bit address for the memory to find out the 8-bit RGB value from the translation table. After TT_ACK gets set to high, the TT_ADDR has been sent to the memory, now the memory is set to the read mode, and the memory sends this 8-bit RGB value out the Dout port. Next, the DACAL_REQ is set high to change the DA for the next entry. The DACAL module is a special counter that follows the usage pattern of the DA described earlier. The software then provides the next lum value and the control sets TT_REQ high. This procedure repeats for two more luminance entries. Then, new C_b and C_r values are provided and the whole process repeats until the entire frame has been processed. After discussing the detail of the datapath, we move to the whole hardware design, which is shown in Figure 7. From this figure, we can see the datapath and the controller are connected together. The major difference between synchronous and asynchronous designs is synchronous design has a clock, and asynchronous design has no clock, so we have to use handshaking signals in between them in order to have the controller and the datapath communicate with each other.

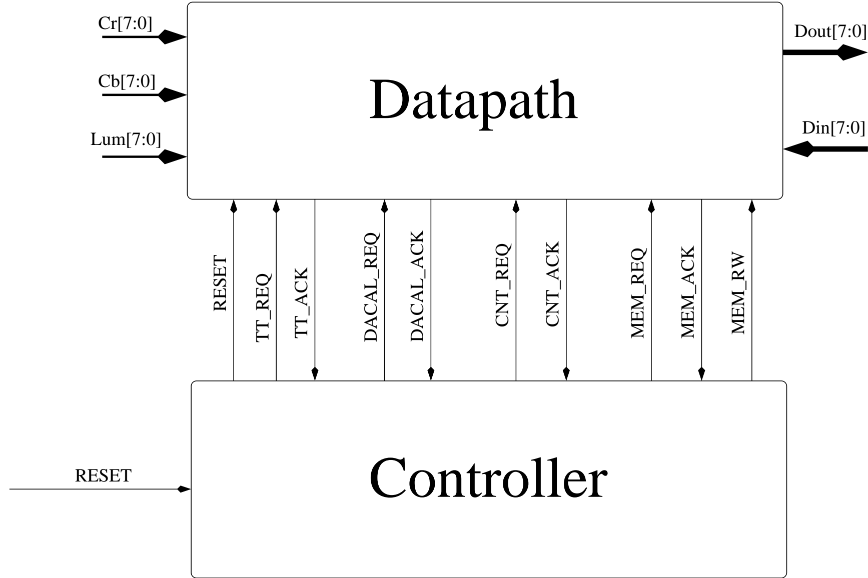

Figure 7: Hardware design.

6 Current Challenges

After going through the whole procedure for this design project, we can see that the design flow requires substantial user intervention. Also, determining and implementing the bundling constraint for the datapath is tricky. It is difficult to design matched delays. Since we are designing an asynchronous ditherer for MPEG-1 decoder, we do need to have some time delay involved in the datapath. However, an FPGA is designed and optimized for synchronous systems, so it does not have direct control to place and route, so the unmanageable net and wire delays might fail in our synthesized designs. It is hard to pick up the right delay time. If we have too much delay time, it hurts the performance of the circuit, if we use too little delay time, it might cause circuit failure. Therefore, how to pick a right number in this range is very problematical, especially the unpredictable place and route makes this problem even harder. We did find out a way to construct a delay line for asynchronous signals, which is to put a number of buffers in between the req and ack signals. We think it is a good method to mimic the behavior of handshaking-level VHDL codes, but like we said before, it is hard to get the right delay time. So far, we have matched delay time for each element, and we think we will make a

main one for the whole datapath when we go to the next step - communication between software and hardware.

Besides those two challenges, after we get both of the software and hardware parts implemented, simulated, and synthesized, we need to put those two parts together to see how they can communicate with each other. This is also a difficult part, because since ATACS is not integrated with the FPGA synthesis tool, hand modifications are still required. Also ATACS standard-logic-based synthesized equations may or may not be implemented by an FPGA because FPGA's physical implementation is very different (PLA or LUT), so we are not sure what kind of optimizations will be applied when we put the ATACS synthesized logic equations through the FPGA synthesis procedure. Because we have already got the datapath synthesized by using WebPack FPGA, we need to get the controller synthesized by WebPack too. Then, the whole design should work, theoretically, if we just connect them together. So we started from the ATACS generated prs file for the controller, using storeVHDL command to make ATACS generate VHDL code. Then, we take this new VHDL code to WebPack, and try to synthesize the code. We have tried several small files from the controller, and they all synthesized successfully. We also tried the whole controller.

7 Conclusion

In this report, we have presented the main idea of this project, which is to design an asynchronous ditherer for an MPEG-1 decoder. After choosing the software algorithm, we partitioned it into a software component and a hardware component. For the software component, we use C++ to implement, for the hardware component, we synthesize and implement onto an FPGA or ASIC. After we get both of those two parts done, we integrate them together to get a mixed software and hardware design. All of the parts have been implemented, simulated, and synthesized successfully. We hope overall it is a more efficient design than the original one. During this design project, we have a better

understanding on channel packages, matched time delay, asynchronous circuit design, and synchronous hardware/software.

In the future, what we would like to do is to automate the entire design flow from high-level channel descriptions to an FPGA. It is hoped that during the course of this research, a better understanding of asynchronous systems will be achieved, and a better complete synthesis method for asynchronous design will be developed.

References

[1] BMRC. Berkeley MPEG Tools. `http://bmrc.berkeley.edu/frame/research/mpeg`.

[2] Neil Gray. Algorithm for C++ MPEG1 Decoder. `http://www.uow.edu.au/~nabg/MPEG/sharfile.txt`.

[3] Neil Gray. Overview for C++ MPEG1 Decoder. `http://www.uow.edu.au/~nabg/MPEG/mpeg1.html#overview`.

[4] Neil Gray. Physical Organization of the Code. `http://www.uow.edu.au/~nabg/MPEG/mpeg1.html#code`.

[5] J.J.Hwang K.R.Rao. *Techniques & Standards for Image. Video & Audio Coding.* Prentice-Hall PTR, Upper Saddle River, New Jersey, 1996.

[6] Chris Myers. *Asynchronous Circuit Design.* John Wiley & Sons, 2001.