Jeremy Stribling, cs150-strib, 13794235
Oliver Zee, cs150-oczee, 13991673
Project Final Report
12/7/01

I. Introduction

For our final project, we implemented a CD player as described in the project specification. The physical components of our CD player include a Xilinx FPGA board, a 16-button keypad, an LCD screen, and a standard ATAPI CD-rom. Our project differs from other projects in several important ways. First, some of the buttons on the keypad have different functions based on the current state of the CD player (for example, there is one button that acts as a stop button, an eject button, and a load button). Second, we output unique symbolic icons to the LCD that show the user the current status of the CD player. Lastly, we added load functionality to our CD player, which allows the user to load a CD with the touch of a button. We made no changes to the project specifications; all specifications were closely followed, with extra features implemented in addition. Our project can be found in cs150-strib\FinalProject\, and the partner's login name is cs150-oczee.

II. Theory of Operation

To use our CD player project, first you must connect a power supply and XChecker cable to the Xilinx board. Next, plug the IDE cable attached to the Xilinx board to the back of the CD-rom, and connect the digital output line of the CD-rom to the Xilinx board. Insert a CD (preferably Core by Stone Temple Pilots) into the CD-rom. Open the Xilinx Hardware Debugger on the computer, and open a new project using our project bit file. Download this onto Xilinx board. Now the CD player is ready for use!

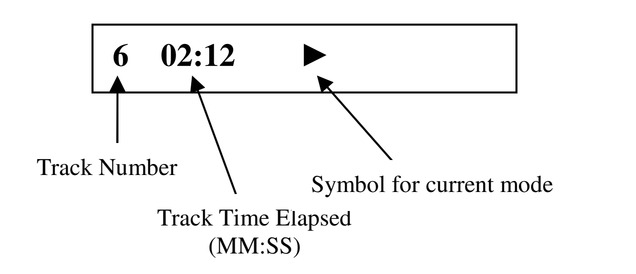

Figure 1: CD Player User Display

The CD player begins in Stop Mode. This is indicated by a black square symbol (■) in the "current mode" position on the LCD (see Figure 1). In this mode, you can start the CD playing at track 1 by pressing button "D" on the keypad (see Figure 2), thereby entering Play Mode, or eject the CD by pressing the “0” button, thereby entering Eject Mode. In Eject Mode (▲), the only function available to the user is pressing the “0” button to load the CD and re-enter Stop Mode. In both stop and eject mode, the track number will read “1” and the elapsed time will read “00:00”.

In Play Mode (►) the user can adjust the volume of each individual speaker, toggle the bass and treble filters for each speaker, jump to the next track or the previous track, and pause the CD (thereby entering Pause Mode). There are 6 volume levels for each speaker. They begin at full volume, and can be adjusted between no volume and full volume using the volume buttons on the keypad (“4” = left volume down, “5” = left volume up, “6” = right volume down, and “B” = right volume up). You can toggle bass (low-pass) and treble (high-pass) on and off for each speaker using the filter toggling buttons (“7” = left bass toggle, “8” = left treble toggle, “9” = right bass toggle, “C” = right treble toggle). The CD we used has twelve tracks, so our CD player is customized to reflect that. You can skip to the previous track by pressing the “*” button, or skip to the next track by pressing the “#” key (this skips circularly between 1 and 12 in hex). By pressing the “D” key in Play Mode, the user can pause the CD and enter Pause Mode. The user can also stop the CD and enter Stop Mode by pressing the “0” key. In Play Mode, the track number on the LCD will reflect the current track, and the elapsed time will show the time the CD player has been playing since the beginning of the current track.

1	**2**	**3**	**A**
4 Left Vol Down	**5** Left Vol Up	**6** Right Vol Down	**B** Right Vol Up
7 Left Bass Toggle	**8** Left Treble Toggle	**9** Right Bass Toggle	**C** Right Treble Toggle
***** Previous Track	**0** Stop/ Eject/ Load	**#** Next Track	**D** Play/ Pause/ Resume

Figure 2: The CD player keypad

In Pause Mode (‖), all of the buttons have exactly the same functionality as in Play Mode except for button “D”, which will cause the CD to resume playing and enter Play Mode. The CD will begin playing at the proper track if next or previous track is pressed. The track number on the LCD will reflect the current track, and the elapsed time will be frozen at what it was when Pause Mode was entered.

III. Control and Datapath Description

We divided the CD player into six top level blocks: a clock divider, which provides the timing clock for the rest of the components; keypad, which models the physical keypad and takes in input from the user; a key translator, which tells the CD player to which mode to transition and to which track to change; LCD, which outputs track number, track time elapsed and current mode symbol to the user; CD driver, which interfaces with the ATAPI CD-rom drive; and a music processor, which takes in a digital bit stream from the CD-rom, processes it according to the current volume, treble, and bass settings, and outputs it to a DAC.

At the top level, when the user presses a key, the keypad block determines which key has been pressed and asserts the keypress signal, which tell the key translator, LCD and CD driver to begin processing the newly pressed key. After the key translator block has computed the new track number and mode, the LCD block will write this information to the LCD screen, and after it is finished, it will assert its done signal. This informs the CD driver block to send the appropriate command to the CD-rom, after which it asserts its own done signal. The keypad block then waits for another key press from the user. The music processor continuously waits for data from the CD-rom and transforms it based on the volume, treble, and bass settings sent to it from the keypad block. See Appendix A.0 for a top-level diagram. Note that the arrows do not represent state transitions, but input and output control signals.

In checkpoint one, we created the keypad, key translator, and LCD blocks. The keypad block continually checks each row of the physical keypad for input, asserts its keypress signal, and outputs the key that was pressed to the key translator. See Appendix A.1, Figure 1 for a diagram of the keypad block. The key translator then computes track and mode and encodes it using one-hot state assignment. See Appendix A.1, Figure 2 for a diagram of the key translator block. The LCD block takes in the mode and track information provided by the key translator and outputs it to the LCD screen. When the design is first downloaded to the Xilinx board, the

LCD block sends initialization commands to the LCD screen. These commands are stored in a ROM and are sent sequentially to the screen. Initially, we could not get the LCD to initialize using the sequence of commands given in the checkpoint one specifications. By using the oscilloscope, we were able to reverse engineer the test bit file provided and discover the correct initialization sequence. See Appendix A.1, Figure 3 for initialization ROM contents and description. After the screen has been initialized, we begin transitioning through a one-hot state scheme. See Appendix A.1, Figure 4 for one-hot state encoding diagram. First it clears the screen, then it does the following operations in sequence: writes the track number, writes a space, writes the minutes of the track time elapsed, writes a colon, writes the seconds of the track time elapsed, and writes the current mode character several spaces to the right. For this checkpoint, since we weren't playing a CD yet, we had a dummy counter that just counted in binary each time a key was pressed in place of the actual track time elapsed. The instructions for writing the special characters are stored in individual ROMs for each character. See Appendix A.1, Figure 5 for a sample special character ROM.

In checkpoint two we added the CD driver block, as well as a real time counter for the LCD screen. We also used a one-hot state schema for the CD driver consisting of the following states: it waits for a keypress, waits for the LCD to be done, sends the ATA packet command to the CD-rom, waits for 100 clock cycles, and sends the six ATAPI packets corresponding to the current mode of the CD player. See Appendix A.2 for a diagram of the CD driver state machine.

Checkpoint three dealt with the data stream coming in from the CD-rom. The bits first go through the sampler, which detects cells to the other blocks and asserts the cell_ready signal. First there is a preamble detector block, which examines the cells and determines whether a preamble has been sent. It alerts the next block, the garbagizer, which throws away the next sixteen cells. When that has been done, the garbagizer alerts the extractor block, which extracts sixteen bits from the next thirty-two cells and sends them all together to sender block. This block reverses the order of the bits that come in from the extractor and sends them to the DAC input pins. See Appendix A.3 for a diagram of this process.

For the last checkpoint, we inserted a DSP block between the extractor and the sender blocks from checkpoint three. This block takes volume, treble, and bass settings from the keypad, and performs arithmetic operations on the data from the extractor as necessary. See Appendix A.4, Figure 1 for a list of input/output signals to the DSP block. It utilizes a RAM to keep track of the last two samples for each channel, as well an accumulator saving intermediate values for its calculations. See Appendix A.4, Figure 2 for a diagram of this RAM. The control portion of the DSP block steps through a forty-five state one-hot machine, which sends control signals to the datapath first for processing volume and then for performing filtering. The datapath consists of the RAM, a 16-bit adder, and a 16-bit shift register. See Appendix A.4, Figure 3, for a description of the register transfer operations performed by the control and datapath.

IV. Design Decisions

For the overall design, we tried to keep all the blocks as modular as possible, in order to keep the design clean and to be able to make changes in one block independent of the other blocks. This implies that each block waits for a control signal to tell it when to start, and asserts a control signal when it is finished. We decided to use one-hot encoding throughout the project, for its simplicity and although it takes more flip-flops than other types of encoding, we felt that it would produce a cleaner design overall, and we had many flip-flops with which to work. Another overall design decision we made was to use logic for the majority of the control portions of our project, as opposed to ROM based implementations, mostly due to the fact that ROMs are significantly larger than logic in terms of CLBs, and logic can be minimized more easily. To resolve timing issues, we designed the project to be a synchronized Mealy machine by registering the outputs of each block.

We made several important design decisions in checkpoint one that had a large impact on the rest of the project. We implemented the keypad block such that it only accepts new key presses after all the other relevant blocks have asserted their done signals, ensuring that the other blocks won't receive a new key press while still processing the previous one. Unlike most of the other blocks, we decided not to implement the key translator block with a done signal. It is unnecessary because while the translator computes mode and track information, the LCD is still clearing the display, and as such, will not examine the input from the key translator until at least one cycle later. This ensures that there will not be any timing problems between the key translator and the LCD, and it saves us several states and signals. In the LCD block, we made several crucial design decisions involving our CLB count. We minimized much of the logic required to send commands out to the LCD screen by extensive use of K-maps. However we did choose a ROM-based command system for the initialization sequence and for sending the commands to write the special characters. In terms of the initialization sequence, it made it much easier to send a sequence of commands and tweak those commands without having to recompute their logic functions. It made sense to use ROMs for the special characters so that more custom characters could be inserted easily just by adding more ROMs without having to perform complicated logic manipulations.

In checkpoint two, we added a real time counter to the LCD block. We created a macro for this counter within the LCD block so that it would be more modular and so that it would output information that was compatible with the logic used in checkpoint one. In our CD driver block, rather than use a ROM-based implementation for our CD's table of contents, we chose to use K-map minimized logic to lower our CLB count. The commands sent to the CD-rom are solely based on which mode the CD player is in. Externally, the user only sees four modes (play, pause, stop, eject) as detailed in section II, but internally we added two more modes: resume and load. The resume mode is identical to the play mode, but since the ATAPI command to resume differs from the command for play, we decided to have a different state for resume. Similarly, the load mode is the same as stop mode.

Modularity helped us simulate and debug checkpoint three. By splitting up the different parts of frame interpretation, we were able to divide and conquer the process, and ensure that each portion worked correctly before combining them. By extensively breaking down the testing, we were able to do all of our debugging in software, and our checkpoint three was successful the very first time we downloaded it to our board. This modularity also helped us for checkpoint four, since we were able to insert the DSP block between two other blocks without any major modifications to checkpoint three. Our sampler block, which detects cells in the bit stream, outputs a control signal called cell_ready, which essentially serves as the clock for the subsequent blocks. This ensures that the other blocks will only look at a cell when it is ready to be processed, eliminating timing issues due to the difference in clock speed between the bit stream and our CD player. We realized that we could halve the number of states that we used in the preamble and extractor blocks by detecting changes in the cell stream, instead of explicit ones and zeros. This is due to the fact that each pattern we are looking has an equivalent pattern that is its inverse (i.e. it goes from one to zero in the same place that its inverse goes from zero to one). Therefore, only the changes are significant, not the actual cells.

In checkpoint four, we decided to use multiplexers instead of tri-state buffers to route signals because they require less control signals, and thus less logic. They may require more CLBs, but this space can be minimized significantly by using LogiBLOX in Xilinx. We also reduced the number of operations required to implement high-pass filtering by adding and then inverting the sum of two numbers, rather than inverting each number individually before adding them. As far as timing was concerned, we fed an inverted clock to the control portion of the DSP block to ensure that the control signals would be stable for the setup and hold times of the datapath components.

V. Evaluation

The following is a description of the utilization statistics for our project, as reported in the Xilinx Implementation Log File:

Device utilization summary:

Number of External IOBs	**46 out of 160**	**75%**
Flops:	**1**	
Latches:	**0**	
Number of CLBs	**324 out of 400**	**81%**
Total Latches:	**0 out of 800**	**0%**
Total CLB Flops:	**318 out of 800**	**39%**

4 input LUTs:	**581 out of 800**	**72%**
3 input LUTs:	**136 out of 400**	**34%**
Number of BUFGLSs	**6 out of 8**	**75%**
Number of TBUFs	**32 out of 880**	**3%**

Our Critical Path was a maximum net delay of 13.746ns.

Our project performs in conformance with the project specifications. The buttons perform the functions described in Figure 2. However, the buttons are not debounced, so the user must press the buttons somewhat quickly or the board will register two keypresses. There are six volume levels for each channel, which range from full volume, as provided by the CD-rom, to zero output. Each channel also has highpass and lowpass filtering, which can be turned on or off using the keypad. The sound coming from the audio jack on the Xilinx board is very clean, without any static whatsoever. In general, our project performs very well, exactly according to the specifications.

VI. Conclusion

We learned a lot from the 140+ hours we spent in Cory on this project. Specifically, we saw how to translate high-level designs and ideas into gate level implementations. Initial design is very important, as we learned by watching ourselves and other struggle with the various checkpoints. We also learned that physical wiring is very, very important, and needs to be handled with care, otherwise you may fry your board, and have to rewire the whole thing over again.

Most of the major problems we encountered involved integration of the different checkpoints and routing issues with the Xilinx software. At the end of checkpoint four, we discovered that the play function from checkpoint two was no longer working, even though it had been fine for the earlier checkpoints, and we had not changed it at all. We resolved this issue by changing the ending time of the ATAPI command to match Mike's (since we were using the same CD). The other major problem we discovered was with the way Xilinx physically implemented our design. We wanted to have dipswitch set up to switch the keypad mode of operation, but when we added the ipad to our design, it induced static in the output, which was totally unrelated to the change. As a result, we had to abandon our extra credit idea.

If we had learned about datapath and control separation earlier, it would have made for a cleaner design if we had used those ideas throughout the project, instead of just in checkpoint four. Our design would have been more user-friendly if we had debounced the keypresses coming from the keypad. We also could have optimized the space utilization by creating our own IO-block in checkpoint two to interface with the CD-rom, because the one provided seemed to have many unused inputs and outputs. And of course, a general Table of Contents implementation would have greatly increased the usefulness of our project.

We only have one suggestion to improve the lab portion of this course. Please do not change the project specifications after they have been released. Make sure there is a working version of the project before handing out the specifications. This has been a very interesting and challenging experience and we will probably remember it until we die.