**Converting Bits to Sound on an FPGA**

**Project members:**

Nicholas Wagner, Deep Patel, Brian Helt,

Hamlet Perkas, Krystal Wang, Abraham Nkum

**Instructor:** Dr. Mohamed El-Hadedy

California State Polytechnic University, Pomona

Electrical and Computer Engineering

29 November 2018

Purpose

Given the instruction that the Verilog project was required to use an implementation of a processor on the FPGA, our group decided to utilize sound as the way to display the processor’s capabilities. This is accomplished through toggling the audio output section of the FPGA after a certain value of time has passed. This creates a square wave. The time between toggles is inversely proportional to the frequency of a note; a shorter time creates higher frequencies while a longer time creates lower frequencies.

The processor is used to receive inputs and store them in memory until they are to be played. Various actions that a traditional processor has such as adding values in memory together can also be done through our implementation. The processor also utilizes 2-dimensional memory in order to store the notes as the note values will almost always be larger than a single bit.

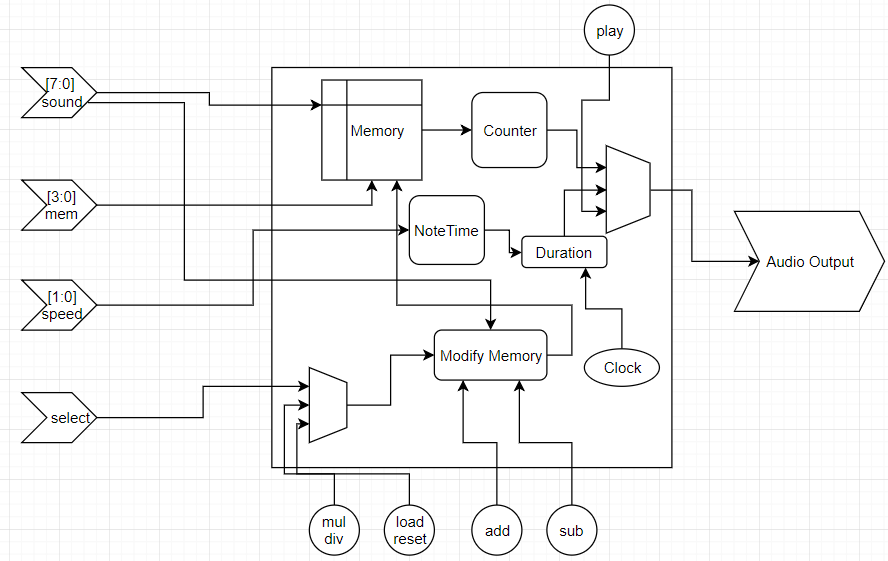
State of the Art

Once the initial idea was chosen, research had to be done to find out where to even start with programming of the FPGA.

1. The site fpga4fun gave the initial idea for how to utilize counters in order to make the FPGA make sound. [1] The site explained how the toggling of the audio output after a counter filled up would be similar to a few hundred hertz signals given the counter incrementation being based off of the FPGA’s clock. We had to modify the code example to be based off of our FPGA clock (100 MHz) and other parameters that will be explained later.
2. This 8 Step Synthesizer from an Instructables.com tutorial gave us the idea for loading and storing notes for later use. [2] We use the same methodology for storing a note in memory at a specific location while assigning it a specific input value. Our input values, however, are basically unrestricted compared to the limited number of predefined values used in the example project.
3. We used the information on a single cycle processor from the site fpga4student.com in order to get an idea for example instructions for a processor to have. [3] Our code implements the instructions in a different manner from the code represented on the site since we use a single module to easily control our processor functions.

Innovation

Block Diagram



The input switches are represented on the left of the diagram. The Sound switches are the 8-bit value that will be stored in memory until it is sent to the counter to be played. Mem switches are the 4-bit address that any modifications to memory will be stored at. These possible modifications to memory are shown by the buttons at the bottom of the diagram. Add takes the value from Sound and adds it to the value stored in memory at the indicated Mem location. Sub, Mul and Div do the same but subtract, multiply and divide instead. Load stores the value at sound in the memory location overwriting the value already stored there. Reset clears that location completely.

The Select switch determines if the primary or secondary function of a button will be chosen. Mul is the primary function of one button while Div is the secondary function. The same methodology is applied to the Load/Reset button. This step had to be included due to hardware limitations. The Nexys 4 DDR is limited to 16 switches and 5 buttons that can be readily reprogrammed. Including a Select switch allows for more than 5 modification instructions to be used without having to resort to external hardware for additional switches and buttons. One Select switch doubles the number of buttons from 5 to 10. Adding a second select switch would increase the button count from 5 to 20. This would allow us to add processor logical functions such as BEQ, AND, and SHIFT to later iterations of this design.

Converting the input values to sound will be covered in this section. By toggling the Audio Output section of the FPGA from 0 to 1 and from 1 to 0, a square wave will be created. Done continuously for a duration of time would create a waveform that would be interpreted as sound to the ear. By using the Clock Frequency as the driver of this toggle, a waveform would be completed every 2 Positive Edges of the Clock. Since the FPGA board used has a 100 MHZ clock, after 10 ns, the output would toggle from 0 to 1 and after an additional 10 ns, the output would toggle from 1 to 0. This process would be at 50 MHZ, much to high of a frequency for the human ear to pick up.

In order to slow down the toggling of the output, the Counter register in the diagram is employed. A counter would be required to fill up completely before the output would be allowed to toggle. If the counter is incremented with the clock as the driver, this would increase the number of clock cycles required before the output would toggle. Instead of 2 Positive Edges of the Clock, the output would toggle from 2 full Counters. Creating a large enough counter would eventually slow the toggle down and subsequently drive the frequency down to the 1000 HZ range which is inside the range that notes used in musical scores are found.

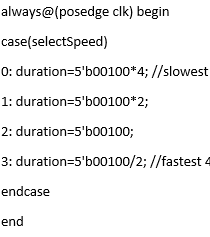
The Sound Switches interact with our formula below to create the counter.



50 MHZ is chosen as the frequency instead of 100 MHZ as it is a slightly lower frequency to get it closer to values needed. Memory[number] is the note taken from the Sound Switches that has been stored in Memory. Number is the memory location of that note stored in memory from 0 to 15. Since our Sound Switches store an 8-bit value, this would set the values from 0 to 255. The \*5 portion of the code increases the range of this 8-bit value from 0 to 1275 while sacrificing some values in between each value. This was done to increase the counter values that we could create. The +9 was arbitrary and was chosen to get our counters modifiers away from the lowest values since we had issues when our counters were very large (only divided by 1 or 2).

Using the example of the formula, an input of 15 is entered. It would be increased to 24 then multiplied by 5 to 120. Dividing 50\_000\_000 by 120 would make 416667 which would be set as the counter value for this specific note. After 416667 clock cycles, the output would toggle from 0 to 1 and after an additional 416667 clock cycles, the output would toggle again. This value would be below 1000 HZ.

The duration of the note is controlled by the Speed Switches and the NoteTime register. This register increments at the same time as the Counter that determines the toggle of the output register; however, it is not reset every time the Counter fills up. It continues to increment until it reaches the value chosen by the Switches. Each of these values below is then multiplied by

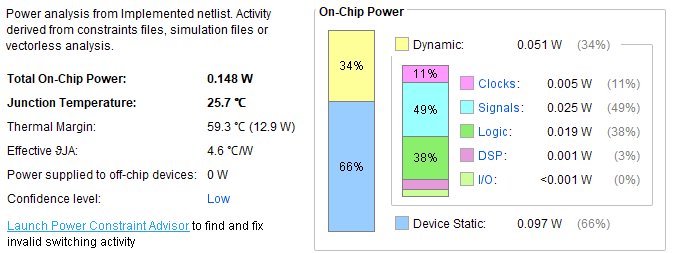


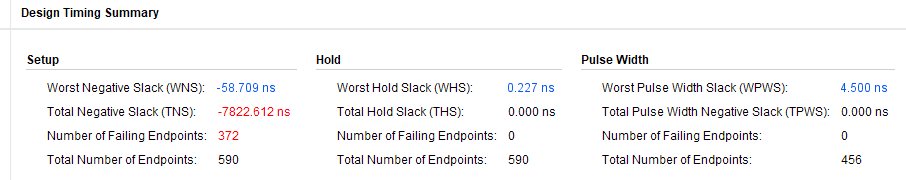


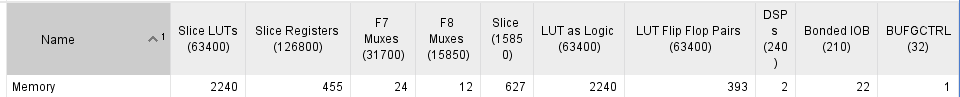
the ClockFrequency to set the number of clock cycles the note should continue to toggle on and off. Setting the multiplier higher will make the note play for a longer time while the inverse would make the note play for a short duration. This NoteTime value is reset every time the Number value changes where number dictates the location of that note in memory. Essentially this means all notes will play for an equal amount of time until the Speed Switches are switched to a different combination.

Utilization

Our project was not made with resource utilization or timing in mind. It uses inefficient options such as using the \* instead of a multiplier module. Many of the registers included in the project are made unnecessarily large thus using more resources than necessary. The subsequent sections in this code will show the resource and timing results of our project.







Conclusion

Our project was successfully created according to the professor’s specifications; however, it was not created with any resource conservation in mind. With additional time, we could remove unnecessary registers and cut down on the size of them. We could also improve the readability by transferring control of specific functions to different modules rather than keeping them inside one module.

Our project has the base laid down to allow us to expand its functionality easily in the future. We could add more processor functions to the already implemented ones to expand its versatility. We could also increase our input note values by working with more advanced input algorithms or by simply adding on external hardware for more switches. More speed options could be added without problem given more switches as well.

An additional option we researched but couldn’t add in an effective way was encryption. Encrypting the input received by the user could be a possible implementation, but we couldn’t figure out a more advanced way than just multiplying the value by a random integer. In addition, encryption in this specific project doesn’t add much to the impact of it. If it was implemented in a larger scale project, encryption may be important.

Contributions

Meetings were held every 2-3 weeks for around 10-20 minutes after class. In these meetings, the overall progress of the project was presented to the group and input was received regarding possible fixes or improvements to the code or implementation on the FPGA. This ensured the project idea was constantly updated and not stagnant from one person’s ideas alone.

Nicholas Wagner did the coding of the project. This was determined from the beginning as it was reasoned that for a project of this scale, it would be easier to have one dedicated programmer rather than trying to coordinate multiple programmers. Ideas for the code were included from most members, however. In addition, he acted as leader for the project and coordinated the overall presentation and report writing and completion. Overall contribution around 35%.

Deep Patel did research on processor capabilities and presented this to the group. This ensured we had an idea for what types of options we should include to modify the memory included in the project. In addition, he did research on encryption when the topic was discussed with the professor. It was determined; however, that we couldn’t find a good way to add this idea without significantly modifying the structure of our project. He attended all meetings and gave his input on the project’s progress. Overall contribution around 20%.

Brian Helt helped in the research of how the project should use the FPGA’s inputs. This eventually led down to the project’s final usage of the switches and buttons. He was also at all meetings and gave input on the project’s progress. Overall contribution around 10%.

Hamlet Perkas helped with preparation for the paper. We didn’t know the scale of the paper until just before the project was due, so this was important for the project. He also attended all meetings and gave input on the project’s progress. Overall contribution around 20%.

Krystal Wang helped create the presentation slides and made sure everyone knew which parts they needed to do. She was also at all meetings and gave input on the project’s progress. Overall contribution around 10%.

Abraham Nkum helped brainstorm through different possible FPGA ideas which eventually led to the group’s decision of choosing sound as our unique project element. He couldn’t attend many of the meetings, however. Overall contribution around 5%.

All members helped with the presentation of the project to the class.

Sources

[1] J. Nicolle, “fpga4fun.com Music Box,” *fpga4fun.com,* [Online].

Available: <https://www.fpga4fun.com/MusicBox1.html> [Accessed 29 November 2018].

[2] L. Tran and J Coplon, “8 Step FPGA Sequencer and Synthesizer,” *instructables.com,* [Online]. Available <https://www.instructables.com/id/8-Step-FPGA-Sequencer-and-Synthesizer/> [Accessed 29 November 2018].

[3] V. L. Le, “Verilog code for 16-bit single cycle MIPS processor,” *FPGA4student.com,* [Online]. Available <https://www.fpga4student.com/2017/01/verilog-code-for-single-cycle-MIPS-processor.html> [Accessed 29 November 2018].

Appendix

module Memory(

input clk,

input [7:0] sound, //input for notes

input [3:0] mem, //input for memory locations where memory will be accessed

input [1:0] selectSpeed, //input to decide which speed note should play at

input play,

input loadorreset, //0 on switchSelect makes this button load, 1=clears a memory location

input add,

input sub,

input multiplyordivide, //a 0 on switchSelect makes this button multiply, 1 = divide

input switchSelect, //when toggled changes what the buttons do to

//let us have 5 more buttons to work with

output reg audio

);

reg [4:0] duration; //Half note

reg [19:0] counter; //Keeps track of how many clock cycles are needed to get the right frequency

reg [29:0] noteTimeCounter; //Keeps track of how long notes should play for

reg [29:0] noteTime; //Holds the duration a note should play for

reg [9:0] number=0; //millisecond counter, and sequence number of musical note.

parameter clockFrequency = 5\_000\_000; //Nexys 100 MHz clock slowed down

reg[19:0] memoryTempLoad;//stores a note from memory temporarily

reg[19:0] memoryTempAdd;//stores a note from memory temporarily

reg[19:0] memoryTempSubtract;//stores a note from memory temporarily

reg[19:0] memoryTempMultiply;

reg[19:0] memoryTempDivide;

reg[19:0] memoryTempInvert;

reg [3:0] stateLoad=0; //state for loading values into memory

reg [3:0] stateAdd=0; //state for adding values together

reg [3:0] stateSubtract=0; //state for subtracting values together

reg [3:0] stateMultiply=0; //state for multiplying values together

reg [3:0] stateDivide=0; //state for dividing values together

reg [3:0] stateReset=0; //state for resetting memory locations

reg [3:0] stateInvert=0; //state for inverting memory locations

reg [19:0] memory [15:0]; //This is 2 dimensional memory

//It stores 16 notes that are up to 20 bits in size

//This block selects the speed that notes will be played at

always@(posedge clk) begin

case(selectSpeed)

0: duration=5'b00100\*4; //slowest 4\*4=16 \* clockFrequency parameter

1: duration=5'b00100\*2;

2: duration=5'b00100;

3: duration=5'b00100/2; //fastest 4/2=2 \* clockFrequency parameter

endcase

end

//This block loads the value on the 8 switches to the memory location specified

always@(posedge clk) begin

if(switchSelect==0) begin

case(stateLoad)

0: begin if(loadorreset==1) begin memoryTempLoad<= sound; stateLoad<=1;end

else stateLoad<=0;

end

1: begin memory[mem]<=memoryTempLoad; stateLoad<=2; end

2: begin if(loadorreset==0) stateLoad<=0; else stateLoad<=2; end

endcase

end

end

//This block adds the value on the 8 switches to the memory location specified

always@(posedge clk) begin

if(switchSelect==0) begin

case(stateAdd)

0: begin if(add==1)

begin memoryTempAdd<=memory[mem]; stateAdd<=1; end

else stateAdd<=0;

end

1: begin memory[mem]<= (memoryTempAdd+sound); stateAdd<=2;end

2: begin if(add==0) stateAdd<=0; else stateAdd<=2; end

endcase

end

end

//This block works like add, but subtracts

always@(posedge clk) begin

if(switchSelect==0) begin

case(stateSubtract)

0: begin if(sub==1)

begin memoryTempSubtract<=memory[mem]; stateSubtract<=1; end

else stateSubtract<=0;

end

1: begin memory[mem]<= (memoryTempSubtract-sound); stateSubtract<=2;end

2: begin if(sub==0) stateSubtract<=0; else stateSubtract<=2; end

endcase

end

end

//This block works like add, but multiplies

always@(posedge clk) begin

if(switchSelect==0) begin

case(stateMultiply)

0: begin if(multiplyordivide==1)

begin memoryTempMultiply<=memory[mem]; stateMultiply<=1; end

else stateMultiply<=0;

end

1: begin memory[mem]<= (memoryTempMultiply\*sound); stateMultiply<=2;end

2: begin if(multiplyordivide==0) stateMultiply<=0; else stateMultiply<=2; end

endcase

end

end

//This block works like add, but divides

always@(posedge clk) begin

if(switchSelect==1) begin

case(stateDivide)

0: begin if(multiplyordivide==1)

begin memoryTempDivide<=memory[mem]; stateDivide<=1; end

else stateDivide<=0;

end

1: begin memory[mem]<= (memoryTempDivide/sound); stateDivide<=2;end

2: begin if(multiplyordivide==0) stateDivide<=0; else stateDivide<=2; end

endcase

end

end

//This block resets the value stored at the specified memory location

always@(posedge clk) begin

if(switchSelect==1) begin

case(stateReset)

0: begin if(loadorreset==1)

begin stateReset<=1; end

else stateReset<=0;

end

1: begin memory[mem]<= 0; stateReset<=2;end

2: begin if(loadorreset==0) stateReset<=0; else stateReset<=2; end

endcase

end

end

always @ (posedge clk) begin

if(play==1) begin

counter <= counter + 1; noteTimeCounter<= noteTimeCounter+1;

if( counter >= (50\_000\_000/((memory[number]+9)\*5)))

//The actual note value is calculated here

//The orginal formula takes half clock speed (50,000,000) then divides by the input

//It was modified to multiply by 5 which reduced accuracy but allowed for

//fewer bits to cover the important range of sounds

//8 bits covers 1-255

//Multiplied by 5 now covers 5-1275

//There were issues with the lower values that couldn't be solved

//Adding 9 to the inputted value removed all problems

begin

counter <=0;

audio <= ~audio ;end//toggle audio output

if( noteTimeCounter >= noteTime) begin

noteTimeCounter <=0; number <= number + 1; end //play next note by incrementing number

end

end

//end

always @(number) noteTime = duration \* clockFrequency; //number of FPGA clock periods in one note.

//this determines how many clock periods a note should play for

endmodule