Youssef Hawas

Omar El-Seadawy

Mostafa Henna

Nour Ghaly

CSCE 2303-02

Digital Design I

Spring 2017

Project III

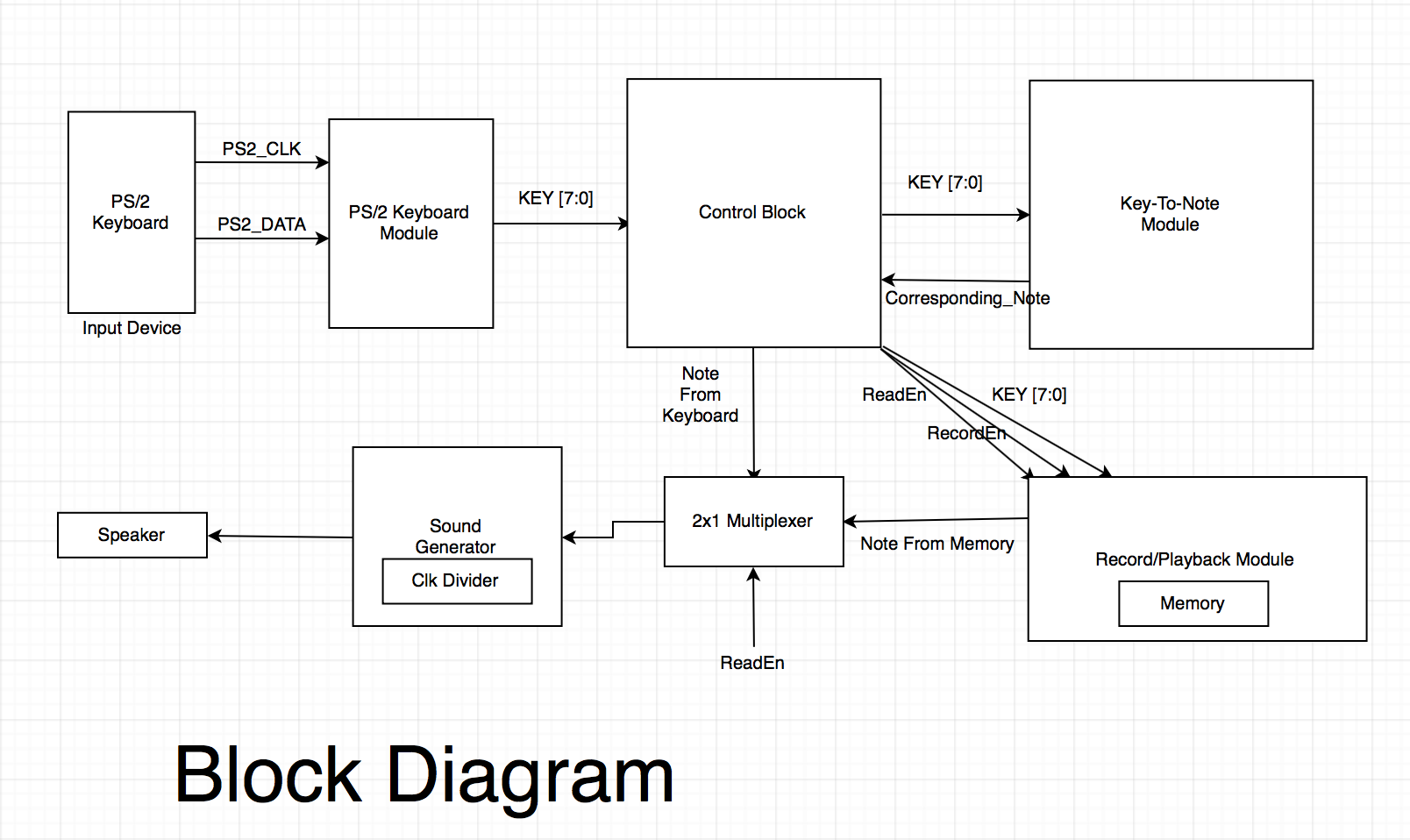
Piano Using PS/2 Keyboard

17/05/2017

**Objective**

The objective of this project was to use all the knowledge we obtained throughout the semester to produce a project which will hopefully impress. We decided to pick option three to go beyond the knowledge of the clock and CPU we made in the lab experiments. Our project allows the user to press on keys using the PS2 keyboard to produce one of the basic piano notes; A, B, C, D, E, F, G using the keys 1,2,3,4,5,6,7 respectively. We also aimed to have a special feature where you could record what you have played.

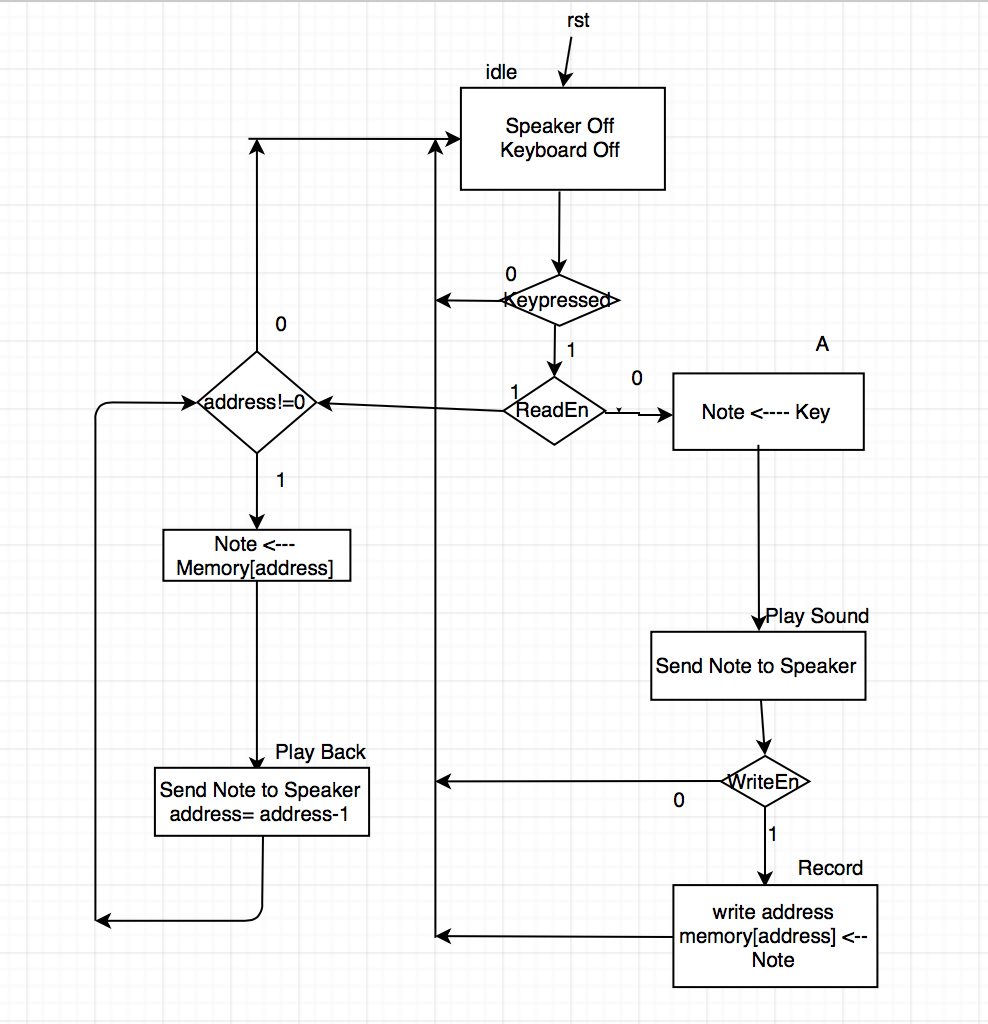
**Block Diagram**



**Dynamics**

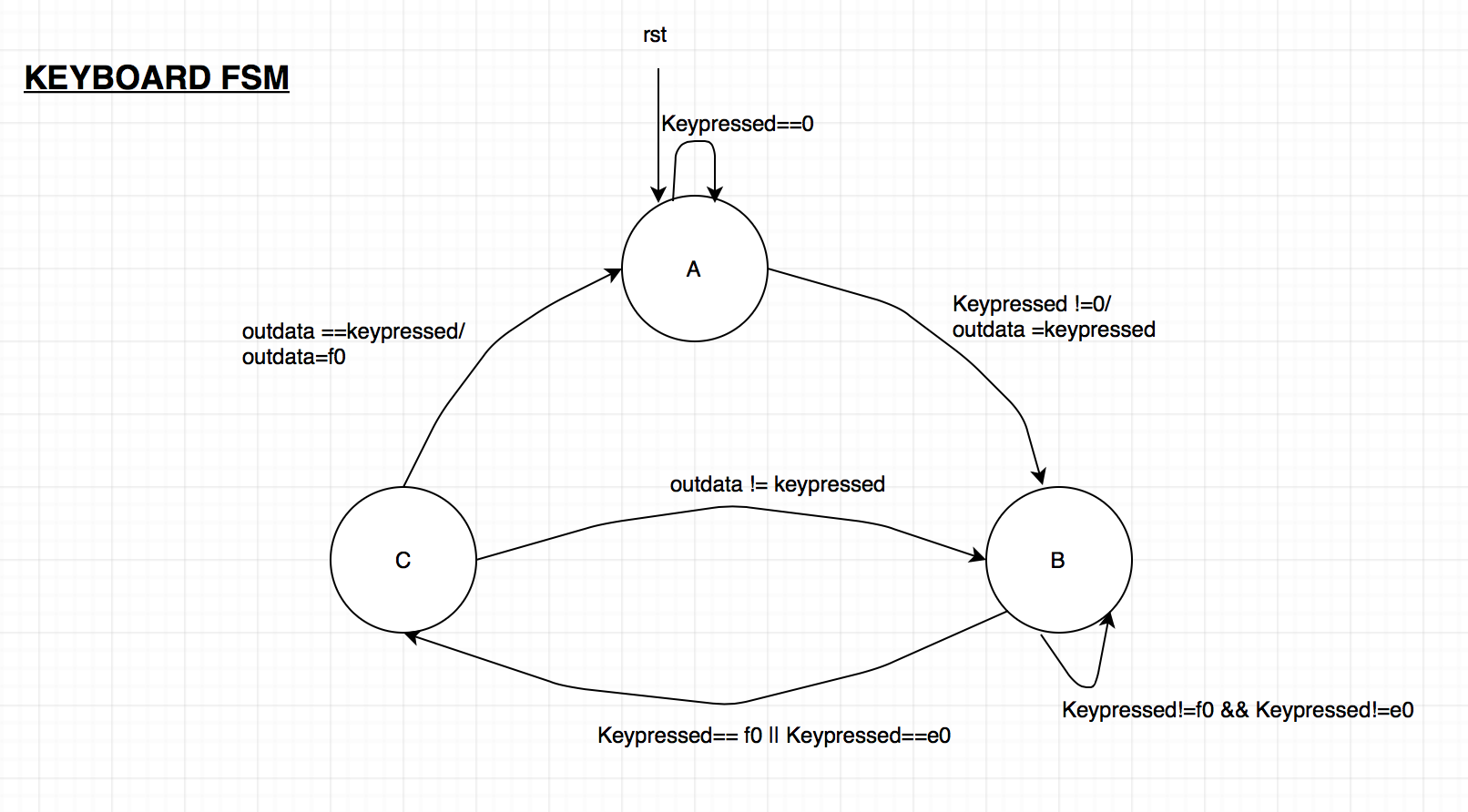
The dynamics of how the code works is simple. When a key is pressed on the keyboard, the keyboard code is generated and sent to a module which compares the key code with the pre-written list of notes and chooses the corresponding frequency value intended to output. This frequency is then sent to a module which generates the sound using a clock divider and the output of clock divider is connected to the speaker in the UCF file. All of this is done using a control unit which is in charge of handling all the addresses and frequencies around our modules.

**ASM Chart**

****

**PS2 Keyboard**

The PS2 keyboard is quite simple, the most difficult thing about it is the fact that it has its own clock. The way it works is that every key has a corresponding ID. So every key pressed, its ID is sent serially as a signal to the FPGA as an address in hexadecimal consisting of eleven bits. The 11 bits are sent over 11 cycles which take around 20-30 KHz, each bit is sent at the negative edge. The first bit is a starting bit which indicates that the upcoming bits are the data that represents the key. The following eight bits are the data bits, followed by one parity bit and end bit to indicate that this is the end of the address. Throughout our code, we only check on the eight data bits and work accordingly.



**Module Keyboard**

This module has two inputs; KDATA and KCLK; and two outputs; OutData, state. Then we have a bus of 8 bits called KeyPressed which stores the data bits of the key and another called ReadCounter. Moreover, there is flag that is used with ReadCounter to indicate that 11 bits are ready to be sent to the KeyToNote module or RecordPlayUnit. We also have parameters which represents our states A, B and C. First, we initialize everything by setting them to 0 (KeyPressed, state, flag), ReadCounter is initialized to one since this reg indicates whether the eleven address bits received from the keyboard are ready or not. Then at every negative edge of KCLK, we increment on the ReadCounter to mark that one bit has been read. The first case statement assigns every incoming bit to the KeyPressed with every increment in the counter. For example, since the first is starting bit, it is ignored and the case starts assigning values from 2. After that, we check if ReadCounter has reached 11; which means that there has been eleven bits read already; we reset it back to one and set the flag to true; otherwise we increment the ReadCounter and flag stays false. The next part is where the data is sent to the KeyToNote module. At the positive edge of every flag, there is a finite state machine that makes sure the data is sent correctly to the other modules. First, it makes sure that there is actual data incoming which is not equal to zero. If incoming data is not equal to zero, it moves to the next state which checks if data is equal to the hexadecimal addresses f0 and e0. We found out that these values are always sent by the keyboard when the key is released. If data is not equal to f0 or e0, which means that the key is still pressed, the output gets the Key code and is sent to the other modules. On the other hand, when data is equal to f0 or e0 it means that the key was released and it moves to state C, which checks if current data is equal to the key code incoming. Coming from the previous condition, if the value is true, the output is assigned f0 in order to silence the speaker and returns to the initial state. If the current data is not equal to key, it returns back to state B in order to re-check the values.

**Module KeyToNote**

This module takes clk, and ReadEn, PBKEY and KEY as inputs and Note as an output. We also have a wire called CheckKey which acts as a 2x1 multiplexer. If ReadEn is 1, it takes the input from the memory and if 0 from the keyboard. Then at the positive edge of the clock, the case statement checks the key that was pressed and assign to Note the corresponding frequency of the key. The default case is that 1 is assigned to Note which means no sound on the speakers. So basically, this module takes the pressed key’s address and based on that, we assign the frequency value of the targeted note to the output Note.

**Module SoundGenerator**

This module is simply a clock divider that takes clk and Note as input that was taken from the KeyToNote module and Sound as output. At the positive edge of the clock, we check whether the counter we have is equal to the Note. If so, we negate the sound and reset the counter to 0. Otherwise, we increment the counter. Sound is sent to the speaker of the Digital Logic Trainer by connecting wires from the FPGA to it.

**Module RecordPlayUnit**

This module takes RPUClk, ReadEn, WriteEn and WKey as inputs and PBKey as an output. Then we have an array of busses as memory and an array of busses to store the time in which the key was pressed (Mem and TimeMem). Firstly, we assign Mem [RAddress] to PBKey to store the address of the note being stored in the memory in PBKey to indicate that this will be the key to playback. Then at the positive edge of the RPUClk, we check if WriteEn is 1, the Address is less than 100 in order not to exceed limit of memory and the current key is the same as the incoming key; we check if the counter is greater than 25000000/8. If those conditions are satisfied, we reset the counter to 0 and increment TimeMem[Address]. Otherwise, we increment the counter. If the current key is not the same as the write key, we increment the address and set Current to the write key. Then we reset the counter back to 0 and Mem [Address + 1] = WKey. Also, in a separate procedural block, at the positive edge of RPUClk we check if ReadEn is 1 (read from memory), and if the address is less than 100 and if TimeMem [RAddress] > an integer d, the integer d is used because we don’t want to change the value inside the memory, so we increment to it instead of decrementing it. Then we check if the counter is greater than 25000000/8; we increment d and reset a separate counter to 0. Otherwise, we increment that counter. If TimeMem [RAddress] < d, we increment the read address (RAddress) and we reset the integer d to 0. Basically, this module checks whether we want to write in the memory or read from it. If we want to write, it checks for the address inputted and we store it in the memory. If we want to read, we check the addresses stored in the memory and sends them to the sound generator module to output on the speaker.