A Modern Music Box

Final Project Report
December 7, 2000
E155

James Speros and Mike Sakasegawa

Abstract:

This project prototypes a digital, programmable music box consisting of a keypad, FPGA, microcontroller, and speaker. The user can select one of twelve notes to play using the keypad. The FPGA debounces and decodes the signal from the keypad and sends it to the microcontroller, which generates a tone code. The tone code is read by the FPGA, which outputs a square wave of the proper frequency to a speaker. Additionally, the user can choose to record and playback songs using the other buttons on the keypad. The sequences of tones are stored on the microcontroller using a sequence descriptor table.

Introduction

Music boxes are well-loved devices that are found in many homes. A typical music box has a rotating cylinder with a specific pattern of bumps. As the cylinder turns, the bumps pluck metal reeds that produce a tone. In this manner they can continuously play pleasant and familiar tunes. However, they suffer from two major shortcomings: only one song can be encoded in an individual box and the duration of this song is typically very short due to physical size constraints. By taking advantage of more modern technology, we have added functionality to overcome these drawbacks.

The system we have designed is a digital music box with record and playback capabilities. To the end user, this system consists of a keypad for input, LEDs to indicate of the system is recording or playing, and a speaker to generate the music. The keypad has 16 keys. Twelve of these keys correspond to notes. The available notes form the major scale spanning over one and a half octaves, starting at C and going up to G in the next octave up. One key starts recording, one starts the playback, and one stops both recording and playback. The final key is inactive.

The system starts up in tone generation mode. It will output tones based on the user's input. If the user presses the RECORD key, the record LED lights up and system records all tone inputs until the STOP button is pressed. Once a sequence has been recorded, pressing the PLAY button will cause the system to play the recorded sequence.

The system consists of the following functional blocks: user interface, polling and debounce, memory, and output control. These functions are realized using a 16-button keypad, a Spartan FPGA, a Motorola 68HC11 microcontroller, a speaker, and two LEDs. The user interface is consists of the keypad for user input, the speaker to play the music and the LEDs for feedback to the user. The polling and debounce module is implemented on the FPGA. The memory module is implemented on the HC11. The output control is split between both the FPGA and the HC11. The FPGA generates the tones that are sent to the speaker, while the HC11 sends control signals to the FPGA that determine which tones will be generated. Additionally, the HC11 controls the LEDs.

Schematics

The schematics for our system include a schematic of the main layout of the circuitry as well as a schematic of the LED circuitry. These schematics show the breadboarded layout, including the keypad, FPGA, LEDs and HC11.

The keypad is the standard 16 button hex keypad with 8 pins. These pins are labeled 1 through 8 from left to right when viewed from underneath with the pins along the top edge. Pins 1, 2, 3, and 7 are the column pins and pins 4, 5, 6, and 8 are the row pins. The column pins are pulled up through 47-kO resistors for the purpose of polling.

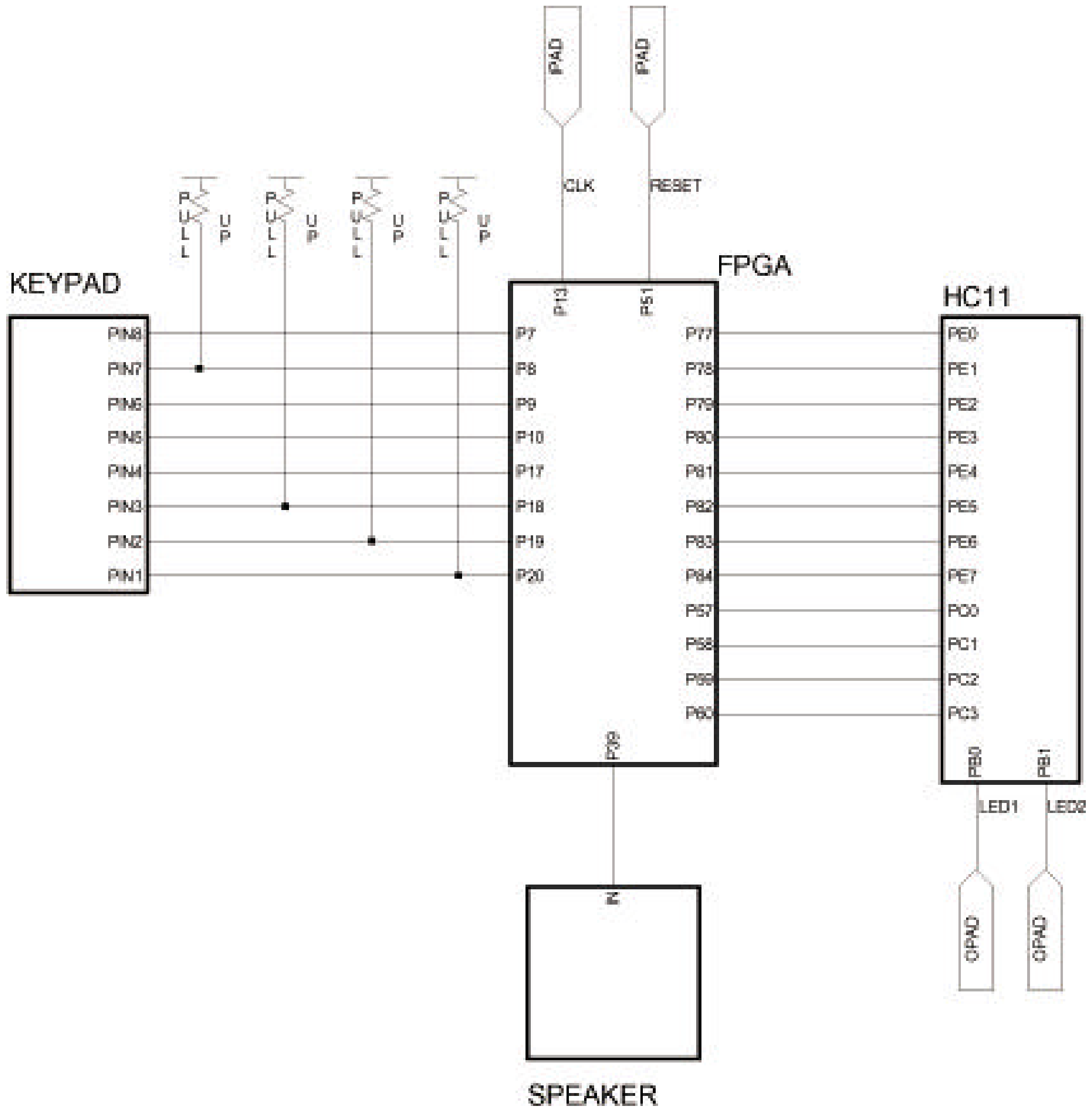

Figure 1: Main Layout Schematic

The output pads, shown on the main schematic as tied to pins PB0 and PB1 of the HC11, drive the LEDs that are used for feedback to the user. These pins drive the bases of two 2N3904 npn transistors through 12-kO current limiting resistors. The emitters of the transistors drive the LEDs through 330-O resistors. The collectors are tied to a 5-V voltage source. When the base is driven high, the path between the emitter and the collector is a short circuit and the LEDs are turned on.

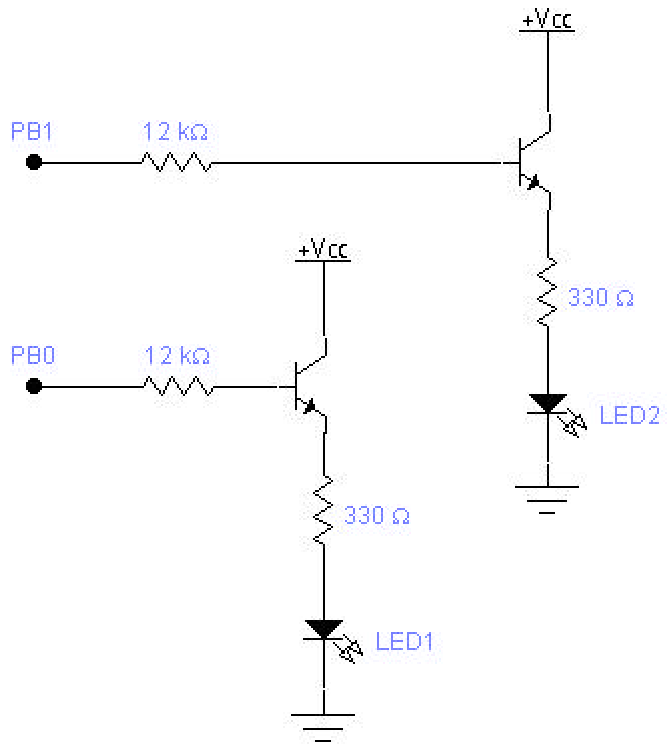

Figure 2: LED circuits

Microcontroller Design

The main functions of the HC11 are memory and output control. The memory is implemented as a sequence descriptor table. The table entries consist of two bytes. The first byte is the tone value to be played, and the second byte is the duration for which the tone is to be played. The output control consists of control over whether the function generation hardware module receives data from memory or directly from the keypad decoder. The code to implement these functions is found in Appendix A.

The HC11 receives one input byte from the FPGA and outputs 4 bits to the FPGA and 2 to the LEDs. The input byte consists of eight bits. From most significant to least significant bit, these are valid button push, stop, play, record, and 4 tone code bits. The four output bits that are sent to the FPGA are tone code bits that are encoded in the same way as the 4 tone code input bits. The two bits that are sent to the LED circuits control whether the LEDs are on or off. The input bits are received over port E, the output to the FPGA is sent over port C, and the output to the LED circuits is sent over port B. All communication is parallel.

The assembly code consists of three main segments: the setup routine, the polling routine, the record routine and the play routine. In the setup routine, the offset index is set. This is necessary because many of the branch commands in HC11 assembly language require indexed addressing. In addition to setting the index, port C is configured as output and both LEDs are set to be initially off.

The polling routine initially checks for any valid button push. When a valid button push is received, it checks to see which button was pressed. If the record button was pushed, the HC11 enters the record routine. If the play button was pushed the HC11 enters the play routine. The stop button is ignored. If a valid button push was detected but neither the record button, the stop button or the play button was pushed, the HC11 sends whatever tone code is currently on the input and outputs it to port C. The entire input byte is actually sent to the port, but only the lower four bits are actually connected to the FPGA. Then the HC11 waits for the button to stop being pushed, at which point it sends a null value to port C. After writing the null value it waits for another valid button push.

The first action in the record routine is lighting the record LED. This LED remains lit as long as the HC11 remains in the record routine. After lighting the LED, the HC11 waits until there is no button being pushed. Then it waits for a button to be pushed. It is important that both conditions be met because recording any initial pause is undesirable. This way, the HC11 will not actually begin recording until the first note is played. After meeting these criteria, the HC11 checks to see which button was pushed. The record and play buttons send the HC11 back to the beginning of the routine. The stop button will send the HC11 to the stop subroutine. If none of these buttons were pressed, then the HC11 enters a subroutine for writing tones. The entire input byte is written to the sequence descriptor table, without actually checking which tone was requested. This way, the HC11 can use the same basic mechanism for outputting tones and pauses. After writing to the table, the HC11 outputs the byte to port C and then measures the duration of the button push. This is done by counting the number of internal timer overflows until the button is released. This gives a resolution of about 30 ms, which is sufficiently small to make almost no audible difference. The number of timer overflows is stored in an accumulator, which can store values of up to 255, or about 7 seconds. If the button is held longer than that, the HC11 writes the maximum value to the table, then restarts the subroutine for writing tones. Since the user is still holding the button down, the next table entry will be the same tone, with no pause recorded in between. Actually, a pause of about 7 clock cycles is produced, but that is more or less inaudible to the ear. If the button is not held down long enough to cause the accumulator to overflow, the duration is written to the table whenever the button is released. After writing the duration, the HC11 enters a subroutine for writing a pause.

The system resolution of about 30 ms is sufficient to assume that there will always be a pause in between two different notes because most people would be unable to switch buttons that fast. Therefore the pause subroutine is always entered after the button is released. The pause subroutine functions almost identically to the tone subroutine. The difference is that instead of writing the input tone to the table, a null value is always sent. Also, the duration is measured from button push to button push instead of from release to release. After this routine is complete, the program re-enters the tone subroutine. Since the tone subroutine always checks for a stop, the record routine can only be exited from there. This is sufficient because any button push will send the HC11 to that subroutine.

The stop subroutine will right a null value as the tone byte and a duration of zero. In this way, the no special mechanism needs to be built into the play routine to handle the tone of a stop. Additionally, the stop subroutine will extinguish the record LED before exiting to the polling routine.

The play routine begins by lighting the play LED. It then reads the first table value, loading the tone into accumulator a and the duration into accumulator b. The HC11 then immediately outputs the tone to port C. Then it checks to see if the duration is zero. If the duration is zero at this point, before any other operations, the current value is a stop, and the play LED is extinguished and the HC11 returns to the polling routine. Since the stop entry always has a null value for the tone byte, this does not produce any sound. If the duration is nonzero, the HC11 checks to see if the stop button has been pressed. If so, the play LED is extinguished and the HC11 returns to the polling routine. Otherwise, it waits for one timer overflow, then decrements the duration accumulator. Then it checks to see if the value in the duration accumulator is zero. If it is, the play routine starts over. Otherwise, the HC11 waits for another timer overflow and then checks again.

FPGA Design

The functions performed on the FPGA include polling and debounce of the keypad and a portion of the output generation. The Verilog code (see Appendix B) consists of four modules to implement these functions. These include the Polling Finite State Machine, the Debounce Finite State Machine, the Function Generator, and the Clock Division circuitry. These four functional modules are instantiated by the Top Level module that handles the interface between the FPGA any other circuitry including the keypad, HC11, and speaker.

The 1MHz clock running the FPGA is too fast for the system because the polling, debounce, and function generation have critical times in the millisecond range and longer. The Clock Division circuitry divides the 1MHz system clock by 2^{12} to produce a slow clock signal with a period of 4.096 ms.

The Polling Finite State Machine determines if a button has been pushed and if so, encodes the keypad information in the signal buttoncode for use by the HC11. All of the column pins on the keypad are pulled up to 5 volts. The Polling FSM holds all of the rows high except for one at any given time. It then checks the columns. If a column line is low, then the button corresponding to the active row and column has been pushed. This information is then encoded is buttoncode. The FSM continues to check the active row until no buttons are pressed in that row.

The Debounce FSM performs two functions. The primary function is rigorous keypad debouncing. When a button is pushed, it verifies a valid push but wasting 8 ms to allow the keypad to stabilize and checking that same button again. (Each clock cycle is about 4ms. There is one state for the button push and one wasted state.) Once it has been verified that a button has been pushed, the valid_push signal is asserted until that button is released. The secondary function of the Debounce FSM is to ensure that valid pushed are not sent to the HC11 to quickly. The HC11 can spend up to about 32 ms executing other tasks between checking the keypad. To avoid missing button transitions, this FSM inserts waste states to ensure that valid_push goes inactive long enough for the HC11 to catch it.

The Function Generator creates the output to the speaker. It produces square waves of a specified period by holding the output, wave, either low or high for half the period and then toggling it and repeating the process. This is performed with a free-running, nine-bit counter. The HC11 sends a four-bit signal (tone) to the Function Generator. This dictates a particular tone to be generated. The Function Generator decodes this into a value equal to half the period (in terms of 4.096 ms clock cycles) of the desired square wave. When the counter matches this half period, it is reset and the wave output is toggled. When no tone is to be generated, the tone input is 0000. Therefore, the tone bits are ORed together to produce an enable signal for toggling the output.

Results

The result of this project was a system that can play notes directly based on user input or simultaneously record and play notes for playback. The system reliably reproduces the user's input. The recorded sequence is audibly identical to the actual user input to a precision of approximately 30 ms.

The record and playback functions were the most difficult part of the design because they were the most complex. The considerations involved include interpreting the output from the polling and debounce, accurately measuring and recording the duration of a note, and consistently updating the function generator. A number of these tasks must be performed in parallel. These parallel tasks needed to be combined into one sequential process that the HC11 could handle. Most of our problems arose from inadequate understanding of the sequence in which to perform the necessary tasks.

There is one major difference between our proposed and actual designs. In our proposal, we outlined a method for generating a sine wave using the HC11 to generate a digital representation of the sine wave that would be interpreted by a digital-to-analog converter and amplified before driving the speaker. Experimentation showed that the pure tone of the sine wave is not aesthetically pleasing. Additionally, the inductive impedance of the speaker made it sensitive to sharp signal edges and relatively insensitive to waveforms without sharp edges. The smooth curve of the sine wave produced low volume even with very high power. Therefore, we drove the speaker using a square wave. The harmonics in the square wave made it sound better. The sharp edges produced a speaker response at an appropriate volume level without any extra amplification. Additionally, using a square wave simplified the digital-to-analog conversion, such that the speaker could be driven directly from the FPGA, rather than using a more complicated circuit.

Driving the speaker directly from the FPGA caused some peculiar behavior. Some measurements made on the system showed that the output voltage swing was from zero to one volt, rather than zero to five volts. We measured the resistance of the speaker at about 8 Ohms. This means that the FPGA was sourcing 125 mA. This is why it couldn't pull the output to an acceptable logic level, but since we weren't using the signal for logic purposes, it didn't matter.

Parts List

Part	**Source**	**Vendor Part #**	**Price**
2N3904 Transistors	Electronics Lab	**-**	**-**
Speaker	Electronics Lab	-	-

Appendices

Appendix A: M68HC11 Assembly Code

```
* This unit passes the note value along and also performs the memory * * functions.
*
* Constant Defines
REGS           equ     $1000
PORTB  equ   $04 * output port for LEDs
PORTC  equ   $03 * output port for tone
DDRC   equ   $07
PORTE          equ     $0A * input port
BIT0           equ     %00000001 * bits 0 - 3 are the tone code
BIT1           equ     %00000010
BIT2           equ     %00000100
BIT3           equ     %00001000
BIT4           equ     %00010000 * bit 4 is the "record" bit
BIT5           equ     %00100000 * bit 5 is the "play" bit
BIT6           equ     %01000000 * bit 6 is the "stop" bit
BIT7           equ     %10000000 * bit 7 is the "button pushed" bit
ZERO           equ     $0
TFLG2          equ     $25
TOF            equ     %10000000
*
* Memory
*
* Main Program
*
* Setup Routine
*
               org     $c000
       ldx   #REGS
* Initialize the data direction of PORTC.
       ldaa  #%11111111
       staa  DDRC,x
* Set the LEDs to the off state initially.
       bclr  PORTB,x BIT1
       bclr  PORTB,x BIT0
*
* Polling Routine
*
* if no button pushed, keep waiting for a button push.
WAIT           brclr   PORTE,x BIT7 WAIT
* if play button pushed, go to play routine.
       brset PORTE,x BIT5 PLAYER
       bra   NEXT
PLAYER jmp   PSTART
* if record button pushed, go to record routine.
NEXT           brset PORTE,x BIT4 RECORD
* if stop button is pushed and not in record routine, wait
* for a new button push.
               brset   PORTE,x BIT6 WAIT
* if a button is pushed and it is not the play, record,
               ldaa    PORTE,x
* or stop button, output the proper tone code to PORTC,
               staa  PORTC,x
* wait until the button is no longer being pushed
BUTTON brset   PORTE,x BIT7 BUTTON
* then output a null value to PORTC
               ldaa    #$0
               staa  PORTC,x
* then wait until next button push.
               bra     WAIT
*
* Record Routine
*
* Set index y as the table pointer and initialize it to the first
* table value.
```

```
RECORD         ldy   #$d000
* Turn on the record LED.
       bset  PORTB,x BIT1
* Wait here for a button push.
RSTART         brset PORTE,x BIT7 RSTART
* Wait here for no button push.
PBUTT          brclr PORTE,x BIT7 PBUTT
* Once you get a button push, check which button.
* If the record button is pushed, wait for another button push.
               brset PORTE,x BIT4 RSTART
* If the play button is pushed, wait for another button push.
               brset PORTE,x BIT5 RSTART
* If the stop button is pushed, to to the stop routine.
               brset PORTE,x BIT6 RECSTOP
* Otherwise the button is a tone button.
* Load the tone code into accumulator a.
               ldaa  PORTE,x
* First store the tone code to the table.
               staa  ZERO,y
* Then output the tone code to PORTC.
               staa  PORTC,x
* Check to see if we are at the last table value.
               cmpy  #$dfff
* If so, go back to the beginning.
               beq   WAIT
* Otherwise, incremement the table pointer.
               iny
* Load accumulator b with the initial duration zero.
               ldab  #$00
* Reset the timer overflow flag.
TOVER          ldaa  #TOF
               staa  TFLG2,x
* Wait here until the timer overflows.
RSPIN          brclr TFLG2,x TOF RSPIN
* Check to see if the duration accumulator has overflowed.
               cmpb  #$ff
* If not, branch to the no counter overflow case.
               bne   NCO
* If there was overflow, store the maximum duration value to the table.
               stab  ZERO,y
* Increment the table pointer.
               iny
* If no button is being pushed, start writing a pause.
       brclr PORTE,x BIT7 PAUSE
* Otherwise start writing whatever tone is being requested.
               bra   PBUTT
* If there was no duration counter overflow, then increment the
* duration accumulator.
NCO            incb
* If the button is still being held down, then go back and wait
* for another timer overflow.
               brset PORTE,x BIT7 TOVER
* Otherwise, write the duration to the table.
               stab  ZERO,y
* And increment the table counter.
               iny
* If no button is being pressed, write a null value to the table.
PAUSE          ldaa  #$00
               staa  ZERO,y
* Output a null value to PORTC.
               staa  PORTC,x
* Increment the table counter.
               iny
* Initialize the duration accumulator to zero.
               ldab  #$00
* Clear the timer overflow flag.
TOVER2         ldaa  #TOF
               staa  TFLG2,x
* Wait here for the timer to overflow.
RSPIN2         brclr TFLG2,x TOF RSPIN2
* Check to see if the duration accumulator has overflowed.
```

```
               cmpb  #$ff
* If not, go to the no counter overflow case.
               bne   NCO2
* Otherwise, write the maximum duration value to the table.
               stab  ZERO,y
* Increment the table counter.
               iny
* If a button is being pushed, start writing the new tone.
       brset PORTE,x BIT7 PBUTT
* Otherwise write another pause.
               bra   PAUSE
* If there was no counter overflow, increment the duration counter.
NCO2           incb
* If there is still no button pressed, wait for another timer overflow.
               brclr PORTE,x BIT7 TOVER2
* Otherwise, write the duration to the table.
               stab  ZERO,y
* Increment the table counter.
               iny
* Start writing the new tone.
               bra   PBUTT
* If a stop value was received, write a null value to the table for
* the tone.
RECSTOP        ldaa  #$00
               staa  ZERO,y
* Increment the table counter.
               iny
* Output a null value to PORTC.
               staa  PORTC,x
* Write a zero duration to the table.
               staa  ZERO,y
* Turn off the record LED.
               bclr  PORTB,x BIT1
* Go back to the beginning.
               jmp   WAIT
*
* Play Routine
*
* go to the beginning of the sequence descriptor table.
PSTART  ldy    #$d000
* Turn on the play LED.
               bset  PORTB,x BIT0
* load the tone code into accumulator a.
PLAY           ldaa    ZERO,y
               iny
* load the duration into accumulator b.
               ldab    ZERO,y
               iny
* output the tone code to PORTB.
               staa  PORTC,x
* check the duration.
               cmpb  #$0
* if the duration is zero, the sequence is over.
               beq     HOME
               bra     PDUR
* If the sequence is over, turn off the play LED.
HOME    bclr  PORTB,x BIT0
* Go back to the beginning.
               jmp   WAIT
PDUR           brset PORTE,x BIT6 PLSTOP
* otherwise reset the timer overflow flag.
        ldaa  #TOF
               staa    TFLG2,x
* wait here until the timer overflows.
PSPIN          brclr   TFLG2,x TOF PSPIN
* decrement the duration.
               decb
* check the remaining duration.
               cmpb  #$0
* if the remaining duration is zero, go to the next table value.
               beq     PLAY
```

```
* otherwise keep waiting.
               bra   PDUR
* If a stop value was received, turn off the play LED.
PLSTOP         bclr  PORTB,x BIT0
* Go back to the beginning.
               jmp   WAIT
```

Appendix B: Verilog Code

```
//Top Level

//Purpose:     This top level is the interface between the FPGA and
//any other circuitry including keypad, the hc11 and the speaker.

module toplevel (clk, reset, column, tone, row, tohc11, wave) ;

        input clk ;
        input reset ;
        input [3:0] column ;
        input [3:0] tone ;
        output [3:0] row ;
        output [7:0] tohc11 ;
        output wave ;

        wire slowclk;
        wire [6:0] buttoncode;
        wire valid_push;

        //module instantiation
        polling_fsm pfsm(column, slowclk, reset, row, buttoncode);

        debounce_fsm dfsm(slowclk, reset, column, valid_push);

        clockdivision cd(clk, reset, slowclk);

        functiongenerator fg(tone, clk, reset, wave);

        //some bit swizzling
        assign tohc11 = { valid_push, buttoncode[6:0] };


endmodule


//Clock Division

//Purpose:     This circuitry divides the clock by 2^12 to produce
//a clock with a period of about 4 ms.  This is necessary because
//input from the keypad can be unstable for up to about 5 ms.
//The debouncing circuitry uses two clock cycles (8 ms) to allow
//the keypad input to stabalize.

module clockdivision (clk, reset, slowclk) ;

        input clk ;
        input reset ;
        output slowclk ;

        reg [11:0] q;

        //counter for slow-down timing
        always @ (posedge clk or posedge reset)
               if (reset)     q <= 12'b0;
               else           q <= q + 1;

        //The slow clock is the highest bit of the counter.
        assign slowclk = q[11];

endmodule


//Polling Finite State Machine

//Purpose:     This state machine polls the keypad.  All of the columns
//on the keypad are pulled up.  The polling fsm holds all of the rows
```

```
//high except for one at any given time.  It then checks the columns.
//If a column line is low, then the button corresponding to the active
//row and column has been pushed.  This is decoded and the signal is
//sent to the hc11 as buttoncode.

module polling_fsm (column, clk, reset, row, buttoncode) ;

input [3:0] column ;
input clk, reset;
output [3:0] row ;
output [6:0] buttoncode ;

reg [2:0] state;
reg [2:0] nextstate;
reg [3:0] row;
reg [6:0] buttoncode;
wire pushbar;

//State values
parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S5 = 3'b101;
parameter S6 = 3'b110;
parameter S7 = 3'b111;

assign pushbar = &column;

// State Register
always @ (posedge clk or posedge reset)
 if (reset) state <= S0;
 else       state <= nextstate;

// Next State Logic
always @ (state or pushbar)
 case (state)

       //Checking row 0
       S0:    nextstate <= S1;
       S1:    if (~pushbar) nextstate <= S0;
              else          nextstate <= S2;

       //Note: Looping back through the states for a given row
       //             allows time for the button push to stabalize
       //             and keeps checking that row until the button is
       //             no longer pushed.  It is possible for the user
       //             to push another button in the row before letting
       //             go of the active button.  This contingency is
       //             handled by the debounce circuitry.

       //Checking row 1
       S2:    nextstate <= S3;
       S3:    if (~pushbar) nextstate <= S2;
              else          nextstate <= S4;

       //Checking row 2
       S4:    nextstate <= S5;
       S5:    if (~pushbar) nextstate <= S4;
              else          nextstate <= S6;

       //Checking row 3
       S6:    nextstate <= S7;
       S7:    if (~pushbar) nextstate <= S6;
       else          nextstate <= S0;

       //Default to handle entering an unknown state
       default: nextstate <= S0;
 endcase
```

```
//State machine output Logic
always @ (state)
 case (state)
  S0:   row <= 4'b0111;
  S1:   row <= 4'b0111;
  S2:   row <= 4'b1011;
  S3:   row <= 4'b1011;
  S4:   row <= 4'b1101;
  S5:   row <= 4'b1101;
  S6:   row <= 4'b1110;
  S7:   row <= 4'b1110;
 endcase

//Keypad decoder logic:  Which button is pushed

/*****************************************************************/
//The encoding of button code is as follows:
//      [3:0]   tone code
//                     0000 = no tone should be generated
//                     0001 = C1
//                     0010 = D1
//                     0011 = E1
//                     0100 = F1
//                     0101 = G1
//                     0110 = A1
//                     0111 = B1
//                     1000 = undefined (never output)
//                     1001 = C2
//                     1010 = D2
//                     1011 = E2
//                     1100 = F2
//                     1101 = G2
//                     1110 = undefined (never output)
//                     1111 = undefined (never output)
//                     note:   if the undefined codes were ever generated, the
//                                    function generator would interpret them as A2
//                                    and generate that tone.
//      [4]            Record button (active high)
//      [5]            Play button (active high)
//      [6]            Stop button (active high)
/*****************************************************************/

always @ (column or row)
 case (row)
  4'b0111:     if (~column[0]) buttoncode = 7'b0000001;//C1
               else if (~column[1]) buttoncode = 7'b0000010;//D1
               else if (~column[2]) buttoncode = 7'b0000011;//E1
               else if (~column[3]) buttoncode = 7'b0000100;//F1
               else buttoncode = 7'b0000000;
  4'b1011:     if (~column[0]) buttoncode = 7'b0000101;//G1
               else if (~column[1]) buttoncode = 7'b0000110;//A1
               else if (~column[2]) buttoncode = 7'b0000111;//B1
               else if (~column[3]) buttoncode = 7'b0001001;//C2
               else buttoncode = 7'b0000000;
  4'b1101:     if (~column[0]) buttoncode = 7'b0001010;//D2
               else if (~column[1]) buttoncode = 7'b0001011;//E2
               else if (~column[2]) buttoncode = 7'b0001100;//F2
               else if (~column[3]) buttoncode = 7'b0001101;//G2
               else buttoncode = 7'b0000000;
  4'b1110:     if (~column[0]) buttoncode = 7'b0010000;//Record
               else if (~column[1]) buttoncode = 7'b0100000;//Play
               else if (~column[2]) buttoncode = 7'b1000000;//Stop
               else if (~column[3]) buttoncode = 7'b0000000;//NO!
               else buttoncode = 7'b0000000;
  default: buttoncode = 7'b0000000;
 endcase

endmodule


//Debounce Finite State Machine
```

```
//Purpose:     This state machine performs two functions.  The
//primary function is rigorous keypad debouncing.  When a
//button is pushed, it verifies a valid push by wasting 8
//ms (Each clock cycle is about 4 ms.  There is one state
//for the button push and one wasted state.) and checking
//that same button again.  If the button is still pushed, a
//valid_push is generated until that button is no longer
//pushed.  The secondary function of the debounce fsm is to
//ensure that valid pushes are not sent to the hc11 to
//quickly.  The worst case scenario is that the hc11 checks
//for valid pushes about every 32 ms.  To avoid missing
//button transitions, this fsm inserts waste states to
//ensure that valid push goes inactive long enough for the
//hc11 to catch it.

module debounce_fsm (clk, reset, column, valid_push) ;

input clk ;
input reset ;
input [3:0] column ;
output valid_push ;

reg [4:0] state;
reg [4:0] nextstate;
reg [3:0] column;

//State values
parameter S0 = 0;
parameter S1 = 1;
parameter S2 = 2;
parameter S3 = 3;
parameter S4 = 4;
parameter S5 = 5;
parameter S6 = 6;
parameter S7 = 7;
parameter S8 = 8;
parameter S9 = 9;
parameter S10 = 10;
parameter S11 = 11;
parameter S12 = 12;
parameter S13 = 13;
parameter S14 = 14;
parameter S15 = 15;
parameter S16 = 16;
parameter WASTE0 = 17;
parameter WASTE1 = 18;
parameter WASTE2 = 19;
parameter WASTE3 = 20;

//State register
always @ (posedge clk or posedge reset)
 if (reset) state <= S0;
 else      state <= nextstate;

//Nextstate logic
always @ (state or column)
       case (state)

              //polling the columns to determine if a button has been pushed
              //The column signals are active low.
              S0:    if             (~column[0]) nextstate <= S1;
                     else if (~column[1]) nextstate <= S5;
                     else if (~column[2]) nextstate <= S9;
                     else if (~column[3]) nextstate <= S13;
                     else nextstate <= S0;

              //states to handle column 0 button push
              S1:    nextstate <= S2;
              S2:    if (~column[0]) nextstate <= S3;//if button is still pressed
generate valid push
```

```
                      else nextstate <= S0;                //else no valid push and wait
for another
               S3:    nextstate <= S4;
               S4:    if (column[0]) nextstate <= WASTE0;  //generating valid push
signal
                      else nextstate <= S4;

               //states to handle column 1 button push
               S5:    nextstate <= S6;
               S6:    if (~column[1]) nextstate <= S7;//if button is still pressed
generate valid push
                      else nextstate <= S0;                //else no valid push and wait
for another
               S7:    nextstate <= S8;
               S8:    if (column[1]) nextstate <= WASTE0;  //generating valid push
signal
                      else nextstate <= S8;

               //states to handle column 2 button push
               S9:    nextstate <= S10;
               S10:   if (~column[2]) nextstate <= S11;//if button is still pressed
generate valid push
                              else nextstate <= S0;                //else no valid push
and wait for another
               S11:   nextstate <= S12;
               S12:   if (column[2]) nextstate <= WASTE0;  //generating valid push
signal
                              else nextstate <= S12;

               //states to handle column 3 button push
               S13:   nextstate <= S14;
               S14:   if (~column[3]) nextstate <= S15;//if button is still pressed
generate valid push
                              else nextstate <= S0;                //else no valid push
and wait for another
               S15:   nextstate <= S16;
               S16:   if (column[3]) nextstate <= WASTE0;  //generating valid push
signal
                              else nextstate <= S16;

               //waste time to ensure that there is a null tone between each valid tone
               WASTE0: nextstate <= WASTE1;
               WASTE1: nextstate <= WASTE2;
               WASTE2: nextstate <= WASTE3;
               WASTE3: nextstate <= S0;//go back to polling the buttons

               //include default to avoid implying latches and to handle errors
               default: nextstate <= S0;
       endcase

//Output logic
//While the state machine is in the fourth state of a particular column cycle, a valid
//button push is being detected, so valid_push is asserted.
assign valid_push = (state == S4) || (state == S8) || (state == S12) || (state == S16);

endmodule


//Function Generator

//The counter counts out the duration to hold a particular voltage level.
//The half-period decoder determines how high the counter has to count to
//to hold a voltage level for half the period of the given tone.  The
//compare-match circuitry compares the value of the counter with the half-
//period decoder.  It then activates the match signal.  When the match
//signal is activated, the out-toggle toggles the output (wave) and the
//counter resets to begin counting another duration.  When the function
//generator is not supposed to be generating a tone, the tone[3:0] input
//is 0000.  Therefore, the tone bits are ORed together to produce an
//enable signal for the out-toggle.
```

```
module functiongenerator (tone, clk, reset, wave) ;

        input [3:0] tone ;
        input clk ;
        input reset ;
        output wave ;

        reg [8:0] q ;
        reg [8:0] compare ;
        reg match ;
        reg wave ;

        //counter
        always @ (posedge clk or posedge reset)
                if (reset)              q <= 0;//asynchronous reset
                else if (match)         q <= 0;//synchronous reset
                else q <= q + 1;

        //half-period decoder
        always @ (tone)
                case (tone)
                        4'b0000:        compare <= 9'b111111111; //no button pushed
                        4'b0001:        compare <= 9'b111011110; //478 cycles-C1
                        4'b0010:        compare <= 9'b110101010; //426 cycles-D1
                        4'b0011:        compare <= 9'b101111011; //379 cycles-E1
                        4'b0100:        compare <= 9'b101100110; //358 cycles-F1
                        4'b0101:        compare <= 9'b100111111; //319 cycles-G1
                        4'b0110:        compare <= 9'b100011100; //284 cycles-A1
                        4'b0111:        compare <= 9'b011111101; //253 cycles-B1
                        4'b1001:        compare <= 9'b011101111; //239 cycles-C2
                        4'b1010:        compare <= 9'b011010101; //213 cycles-D2
                        4'b1011:        compare <= 9'b010111110; //190 cycles-E2
                        4'b1100:        compare <= 9'b010110011; //179 cycles-F2
                        4'b1101:        compare <= 9'b010011111; //159 cycles-G2
                        default:        compare <= 9'b010001110; //142 cycles-A2 (shouldn't
happen)
                endcase

        //compare-match circuitry
        always @ (q or compare)
                if (q == compare)       match <= 1;
                else                            match <= 0;

        //out-toggle
        always @ (posedge clk or posedge reset)
                if (reset)              wave <= 0;

                //The following test is performed to verify that a tone should be created.
                //All valid tones have one or more 1's in their encoding.
                else if (tone[0] | tone[1] | tone[2] | tone[3])
                        if (match)      wave <= ~wave;

endmodule
```