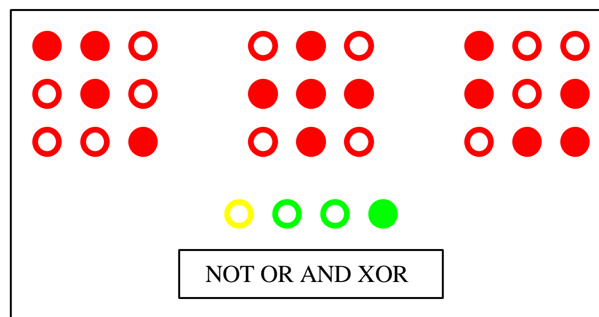
Digital Logic Game

Final Project Report
December 9, 2000
E-155

Michael James Messina and Richard Trinh



Abstract:

Beginning computer engineering students occasionally have difficulty grasping the concepts of digital logic. Standard digital logic exercises can become tiresome and repetitive, so a fun, new method of teaching digital logic could help students understand. This project prototypes a digital logic game consisting of an array of LEDs, hexadecimal score display, a keypad, a microcontroller, and FPGA. Two patterns and one of six logic functions are pseudo-randomly selected. The player has some amount of time to perform that function with the two patterns and enter it with the keypad. The FPGA determines if the answer is correct, and adjusts the score and difficulty appropriately. The player wins when he correctly answers ten patterns.

Introduction

The team has designed and built a game displayed on an array of LEDs that teaches the player about digital logic. The player is shown two pseudo-randomly generated patterns of lit LEDs on a 3×3 grid. The player is also shown a set of four LEDs that corresponds to six possible pseudo-randomly generated digital logic operations: OR, AND, XOR, NOR, NAND, XNOR. The player then tries to perform the selected digital logic operation on the two given patterns, then inputs the result onto a third 3×3 grid using a keypad. The player has some amount of time to complete and submit the answer pattern, and this amount of time is based on the player's current score. The remaining time available to complete the pattern is displayed as a binary countdown. The player earns one point for each correctly answered pattern, and wins when he has a score of A.

A functional block diagram is shown in Figure 1. The HC11 pseudo-randomly generates two patterns and chooses a logic function. The FPGA polls the keypad for a key press, toggles the corresponding LED in the grid, determines the correctness of answers and score, outputs the game data to the LED array and the score to the hexadecimal display, and determines when the player wins the game.

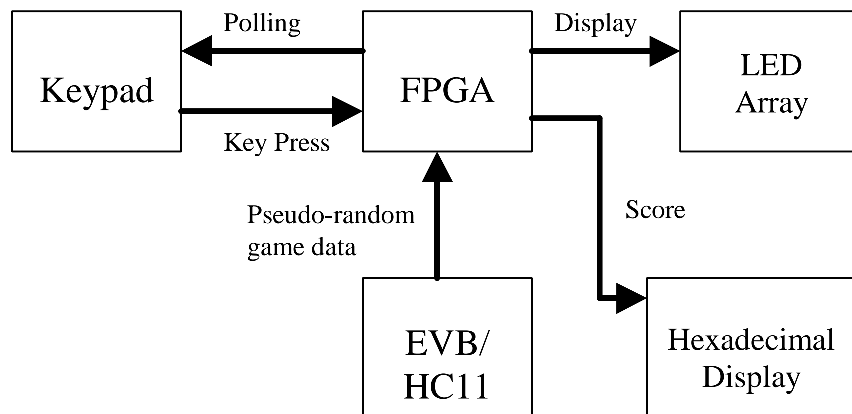


Figure 1: Functional Block Diagram

Schematics

Below are the final breadboarded schematics. Figure 2 shows the schematics for the keypad, Figure 3 for the hexadecimal display, Figure 4 for the pattern LED array, and Figure 5 for the logic function LEDs.

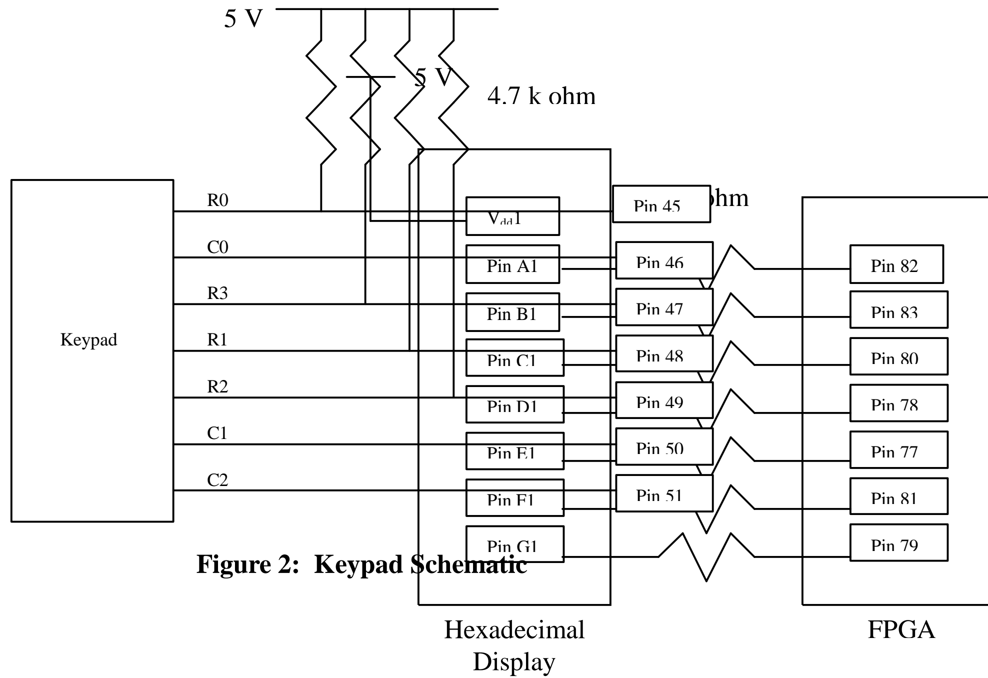


Figure 2: Keypad Schematic

Figure 3: Hexadecimal Display Schematic

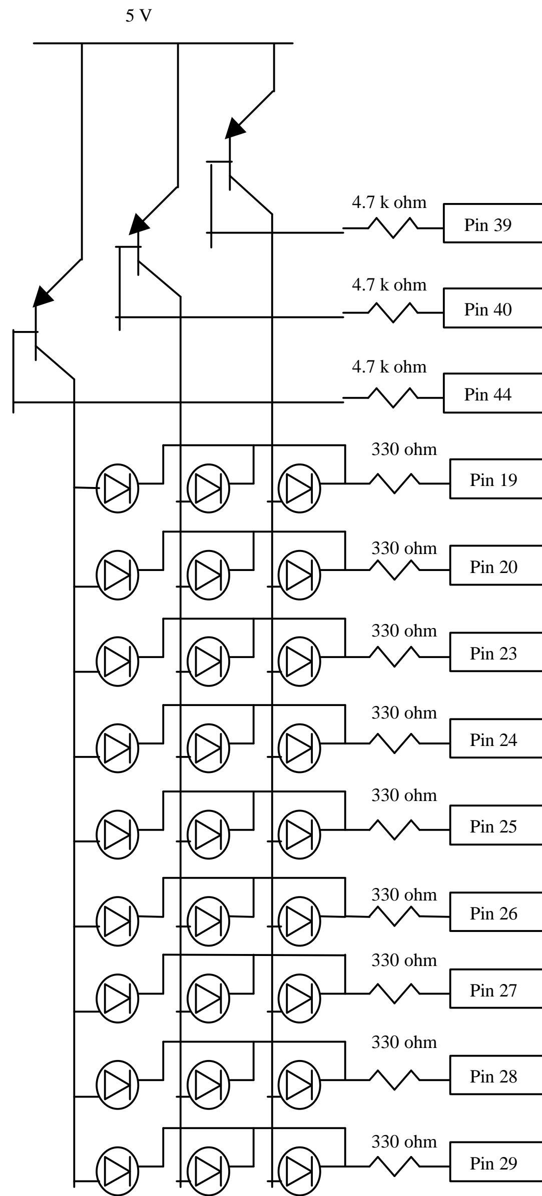


Figure 4: LED Array Schematic

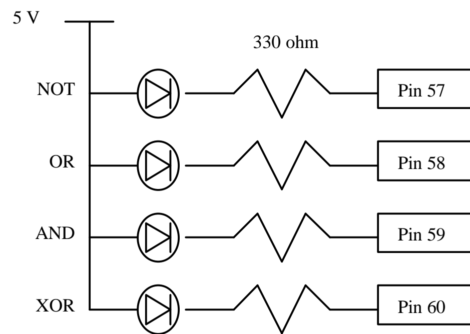


Figure 5: Logic Function LED Schematic

Microcontroller Design

An HC11 was used to generate pseudo-random patterns for the game. The input to the system was the Port A zero pin (PA0). The HC11 polls this pin in a simple loop. When the pin is high, a pattern is generated, which is discussed below. This pin is connected to the FPGA output "GETPAT" and is high when the S0 state in the FPGA is reached (see FPGA Design). The patterns are then generated.

The algorithm used followed the idea of a linear shift register implemented in assembly. A pseudo-random pattern of 31 bits needed to be generated, since at least 21 bits of pattern were needed for the game. This required a seed-value of 5 bits long. According to a web page written by Clive Maxfield, the "taps" for such a register are the are the second and the last bits. The algorithm in assembly was implemented as follows:

1. Use the last 5 bits of the timer ($100E) as a seed value and put it in an accumulator.
2. Determine bit 5 (using 7:0 convention) by an XOR operation of bits 0 and 3.
3. Shift the accumulator right by one bit.
4. Place bit 0 into the next bit of the pattern bytes in memory ($D100-$D103)
5. Perform Steps 2-5 as needed until 32 bits are determined.

After the pattern generation, the Serial Peripheral Interface (SPI) is set up for serial data transfer to the FPGA. The HC11 is set up as the master, and thus drives the transfer using SCK and uses the MOSI pin to hold each bit of data. These pins are outputs from the HC11 and connect to the FPGA inputs "SCK" and "MOSI" (see FPGA Design). Once the pattern is sent, the HC11 begins polling PA0 until the next GETPAT signal from the FPGA is sent.

FPGA Design

The team has written Verilog code to control the game LEDs and the keypad. There are twelve Verilog modules: *main*, *FSM*, *counter*, *scorekeep*, *countdown*, *keypad*, *anspatterntoggle*, *display*, *clkdiv*, *determinecorrectpattern*, *shiftreg*, and *segdisp*. The timing of the game are controlled with a finite state machine in the module *FSM*. Other game functions, such as providing a countdown timer, determining the correct answer pattern, keeping score, and displaying the score are controlled with the modules *countdown*, *determinecorrectpattern*, *scorekeep*, and *segdisp*, respectively. The module *display* controls the 31 game LEDs, and the modules *keypad* and *anspatterntoggle* control the function of the keypad. The module *clkdiv* slows the clock by 4096 times to prevent blending in the multiplexing of the LED display. The module *shiftreg* processes the pseudo-random input from the HC11 which helps to create the game patterns and logic function. The module *main* unifies the other modules.

Controlling the 27 pattern LEDs can be divided into two distinct tasks. The first task consists of controlling the two patterns given to the player. These patterns are created using a pseudo-random pattern generator via the HC11. The second task of controlling the LEDs consists of controlling the 9 LEDs of the answer pattern whose input is determined by input from the keypad. The 27 pattern LEDs currently use three PNP transistors to multiplex in a 3×9 array, requiring 12 pins on the FPGA to drive them.

The *keypad* module determines when a key is pressed through the method of polling, a technique used to reduce the number of pins required on the FPGA and to reduce the amount of logic required for a keypad with a discrete switch for each button on the keypad. For our game, we use buttons on 4 rows and 3 columns of the keypad, requiring 7 pins on the FPGA. Once key presses are recognized, the *anspatterntoggle* module interprets them and toggles the 9 LEDs of the

answer pattern between being lit and unlit. The layout of the keypad is shown in Figure 6. The keys labeled T toggle the LEDs for the answer pattern, the key labeled S submits the displayed answer pattern, the key labeled N requests the next pattern to continue the game, and the black keys are unused.

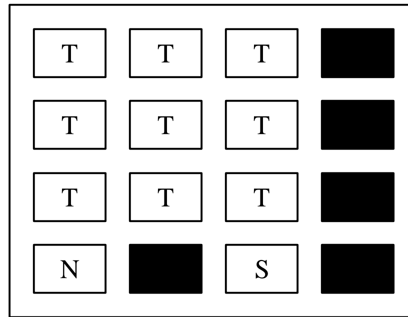


Figure 6: Keypad Layout

The finite state machine has four states, S0 through S3. The state transition diagram is shown in Figure 7. In S0, the program waits for a pseudo-random pattern and remains in S0 until one is received. Once received, the pattern is sent to the display and the program moves to S1, where it waits for input from the player. It is in this state that the player can toggle and submit the LEDs of the answer pattern. Once the player submits or the countdown timer reaches zero without a submission, the program moves to S2, where it blinks the correct answer in the LEDs of the answer pattern. During the transition from S1 to S2, the program also increments the score and adjusts the countdown timer if the player submitted the correct pattern. Each time the player scores, the countdown for the next pattern is reduced by four seconds, effectively making the game progressively more difficult, ranging from 56 to 20 seconds. The program remains in S2 until the player requests the next pattern by pressing the N button, upon which the program returns to S0. In order for the game to not start immediately, the program moves to S2 upon reset, effectively

making the player press the N button to begin the game. When the player correctly answers ten patterns, the program asserts win, the hexadecimal displays an A, and the LEDs display + s. At this point, when the player requests the next pattern, none is given and the program moves to S3 where it remains until reset.

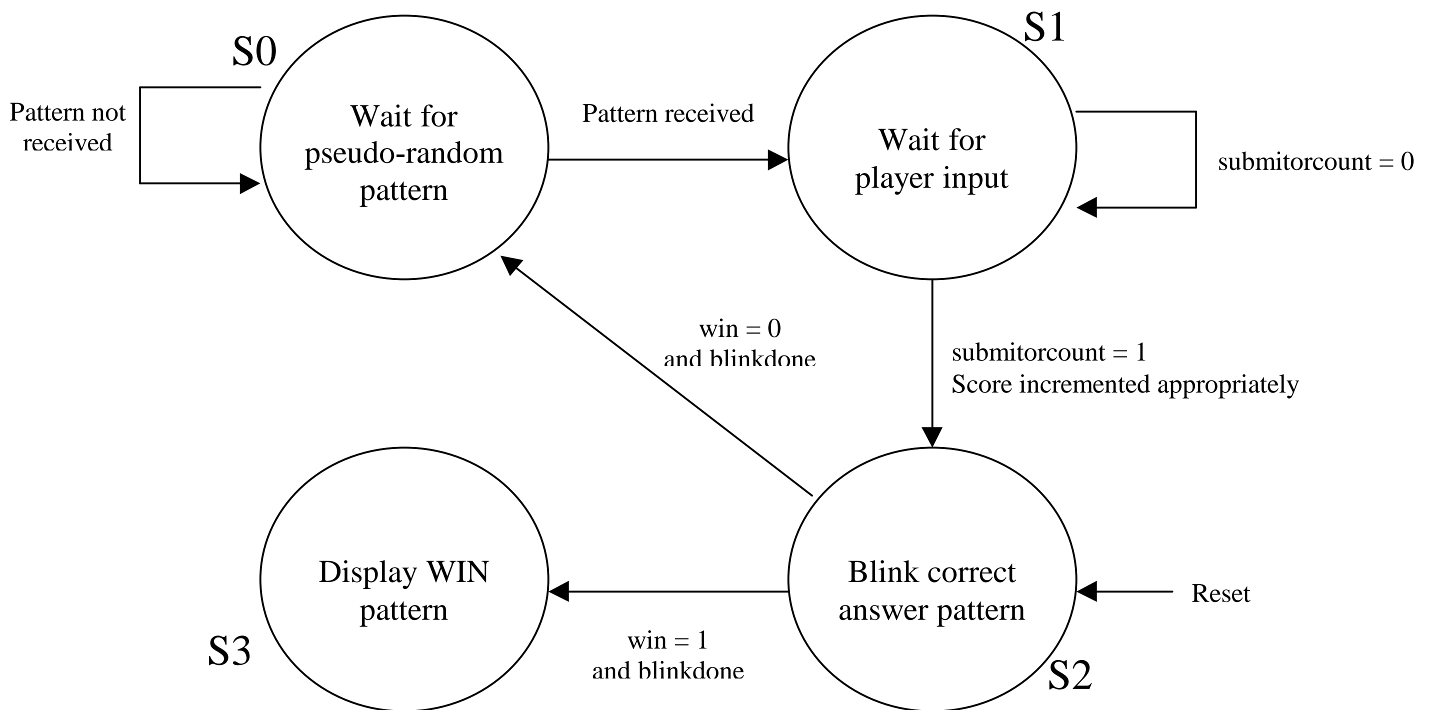


Figure 7: State Transition Diagram

Results

At the time of project checkoff, the game functioned correctly when supplied with a pre-determined pattern, but could not generate a pseudo-random pattern because of two major problems with our implementation. The first problem was that the enabling bit of the SPI initialization was simply not enabled. That caused the SPI wires to simply float and no data to be sent to the FPGA. After the code was modified to enable SPI, the patterns were sent as expected.

The second major problem was that the HC11 locked after generating pseudo-random numbers after a few seconds. This was caused when the first bit of each random byte was written. The first bit required no shifts, so this was taken care of after the label FIRST. Previous to this, Y was pushed on the stack to keep track of what bit in the random byte was to be writen. Y could then be used to count down the number of shifts. In the code after the FIRST label, however, the code failed to pull Y back off the stack, and every time the first bit of a byte was written, the stack increased. This eventually caused a stack overflow and the HC11 froze. The problem was solved by simply pulling Y from the stack at the proper time.

After these problems were diagnosed and corrected, the team re-implemented the design. The game now generates two pseudo-random patterns and selects a logic function, toggles the LEDs of the answer pattern as determined by input from the keypad, appropriately increments the score, displays the correct answer, and moves to a win state when the score is A. A problem does exist however. It seems that a non-negligible amount of the patterns are all zeros. This problem does not appear to be in the HC11 code, so it is probably in the FPGA code. This problem was not fixed, but its effect on the game are not serious.

Overall, the team is satisfied that the game performs according to the original proposal, with a few minor changes. The team originally intended for the score to decrement when an incorrect

answer was submitted or when the countdown timer reached zero. Additionally, the score would start at 7 and the game would end when the score reached 0 or F. The scoring paradigm is arbitrary, and the team later felt that the player should not lose points. Therefore, the scoring system was changed so that it started at 0 and ended at A. Another change to the original proposal was that after connecting the circuitry and testing the keypad, the team determined that the key presses did not need to be debounced, so the team no longer pursued that feature.

Considerations for future work on this project include implementing additional game features, such as sound or a larger array of LEDs.

References

[1] Clive Maxfield. "The Ouroboros of the digital consciousness: linear-feedback-shift registers". *EDN Access*. January 4,1996. http://www.ednmag.com/ednmag/reg/1996/010496/01df4.htm

[2] Motorola. M68HC11 Reference Manual, Revision 3.1991.

Parts List

The project required no components from outside the Microprocessors Laboratory.

Parts required include:

? 27 diffuse red LEDs, 3 diffuse green LEDs, 1 diffuse yellow LED

? 1 keypad

? 1 hexadecimal seven-segment display

? 1 FPGA board

? 1 HC11 evaluation board

? 1 perforated circuit board

? 1 breadboard

Appendices

1. Microcontroller Code

```
0001                          *Code by Michael James Messina
0002                          *December 4, 2000
0003                          *This code generate pattern when  portA[0] is high
0004
0005 d104                     RBYTE   EQU $D104     RBYTE holds running byte in shift   reg
0006 ffb2                     OUTLHF  EQU $FFB2     these next three are for debugging
0007 ffb5                     OUTRHF  EQU $FFB5
0008 ffc4                     CRLF    EQU $FFC4
0009 100e                     TIME    EQU $100E     our seed  rbyte will be the timer
0010 1009                     DDRD    EQU $1009     PORT D DATA DIRECTION
0011 1028                     SPCR    EQU $1028     SPI CONTROL
0012 1029                     SPSR    EQU $1029     SPI CONTROL
0013 102a                     SPDR    EQU $102A     SPI DATA
0014 1000                     PORTA   EQU $1000     Port A will be used to poll for signal from FPGA
0015
0016 d000                             ORG $D000
0017 d000 b6 10 00            POLL    LDAA PORTA
0018 d003 84 01                     ANDA #$01
0019 d005 81 00                     CMPA #$00
0020 d007 27 f7                     BEQ POLL        if  porta[0]=0 then go back to POLL
0021
0022 d009 86 00                     LDAA #$00
0023 d00b ce d1 00                  LDX #$D100      X will hold which p-random byte we are filling
0024 d00e 18 ce 00 00               LDY #$0000      Y will hold the offset within that byte
0025 d012 86 00                     LDAA #$00       Clear all random bytes
0026 d014 b7 d1 01                  STAA $D101      We don't want repeated patterns
0027 d017 b7 d1 00                  STAA $D100
0028 d01a b7 d1 02                  STAA $D102
0029 d01d b7 d1 03                  STAA $D103
0030 d020 b6 10 0e                  LDAA TIME       Timer gives seed 5 bit number
0031 d023 84 1f                     ANDA #$1F
0032 d025 b7 d1 04                  STAA RBYTE      Called running  byte really running 5bits
0033 d028 b7 d1 04                  STAA $D104      For debugging purpose (check random numbers)
0034 d02b b6 d1 04                  NEXTRB  LDAA RBYTE
0035 d02e 84 01                     ANDA #$01       Take the last bit of the running byte (RBYT E)
0036 d030 f6 d1 04                  LDAB RBYTE
0037 d033 c4 08                     ANDB #$08       and bit 3 (7:0)
0038 d035 57                        ASRB            move bit to bit 0
0039 d036 57                        ASRB
0040 d037 57                        ASRB
0041 d038 1b                        ABA             the 0th bit will be the XOR of the bits
0042 d039 84 01                     ANDA #$01       this bit will be the bit 5 (put it
0043 d03b 48                        LSLA            in bit 6 and then shift entire byte)
0044 d03c 48                        LSLA
0045 d03d 48                        LSLA
0046 d03e 48                        LSLA
0047 d03f 48                        LSLA
0048 d040 ba d1 04                  ORA  RBYTE      add the bit in
0049 d043 47                        ASRA            shift byte right to  get next running byte
0050 d044 b7 d1 04                  STAA RBYTE
0051 d047 84 01                     ANDA #$01       Now we want new 0th bit to be added to random byte
0052 d049 18 3c                     PSHY            Save current Y in order to use Y as a count for s  hifting
0053 d04b 18 8c 00 00               CPY  #$0000     If Y=0 then no shift is necessary
0054 d04f 27 24                     BEQ  FIRST
0055 d051 48                        SHIFT   LSLA            Otherwise there is at least one shift
0056 d052 18 09                     DEY             Decrement the number of shifts
0057 d054 18 8c 00 00               CPY  #$0000     See if 0 shifts left
0058 d058 26 f7                     BNE  SHIFT      if not shift again
0059 d05a aa 00                     ORA  0,X        otherwise add the bit to the p -random byte in proper location
0060 d05c a7 00                     STAA 0,X
0061 d05e 18 38                     PULY            Restore Y
0062 d060 18 08                     INY             Increment it (next bit requires 1 more shift)
0063 d062 18 8c 00 08               CPY  #$0008     8 shifts => we are on next p-random byte
0064 d066 26 0a                     BNE  SAMEB      otherwise we are on the same byte
0065 d068 18 ce 00 00               LDY  #$0000     if on next byte #shifts=0
0066 d06c 08                        INX             let X p oint to next p-random byte
0067 d06d 8c d1 04                  CPX  #$D104     if X>d103 we have a 32 bit p-random number in [d100:d103]
0068 d070 27 0e                     BEQ  DONE       and we are done
0069 d072 7e d0 2b                  SAMEB   JMP  NEXTRB     otherwise  calculate next running byte
0070
0071 d075 aa 00                     FIRST   ORA  0,X        set the first bit of the current RNDBYTE
0072 d077 a7 00                     STAA 0,X
0073 d079 18 38                     PULY            restore Y
0074 d07b 18 08                     INY             next bit will have to be shifted once
```

```
0075 d07d 7e d0  2b                  JMP  NEXTRB     calculate next running byte
0076 d080 b6 d1 00            DONE    LDAA $D100
0077                           *        TAB
0078                           *        JSR OUTLHF
0079                           *        TBA
0080                           *        JSR OUTRHF
0081                           *        JSR CRLF
0082                           *        LDAA $D101
0083                           *        TAB
0084                           *        JSR OUTLHF
0085                           *        TBA
0086                           *        JSR OUTRHF
0087                           *        JSR CRLF
0088                           *        LDAA $D102
0089                           *        TAB
0090                           *        JSR OUTLHF
0091                           *        TBA
0092                           *        JSR OUTRHF
0093                           *        JSR CRLF
0094                           *        LDAA $D103
0095                           *        TAB
0096                           *        JSR OUTLHF
0097                           *        TBA
0098                           *        JSR OUTRHF
0099                           *        JSR CRLF
0100                           *        JMP POLL
0101                          *************************************
0102                          * SPI Initialization                 *
0103                          *Info taken from Lab 6 solutions     *
0104                          *************************************
0105                          *Configures the SPI:
0106                          *Bits 7,6: Always 0
0107                          *Bit 5: 1, Slave Select=>ignore   ssbar
0108                          *Bit 4: 1, SCK, =>master
0109                          *Bit 3: 1, MOSI, =>master
0110                          *Bit 2: 0, MISO, ignore for master
0111                          *Bit 1,0: 0, SCI,  because ?????
0112                          *************************************
0113 d083 86 38                      LDAA #%00111000
0114 d085 b7 10 09                   STAA DDRD
0115                          *************************************
0116                          *SPI  Initialization  Continued       *
0117                          *************************************
0118                          *Bit 7: SPIE (SPI interrupt enable)=0 (disable interrupts)
0119                          *Bit 6: SPE  (SPI enable)=1 (enable SPI)
0120                          *Bit 5: DWOM ( PortD wired-OR)=0 (Disable DWOM)
0121                          *Bit 4: MSTR (SPI Master Mode)=1 (SPI is master)
0122                          *Bit 3,2 CPOL ,CPHA (clock polarity and phase)=11 for SCK active on rise
0123                          *Bit 1,0 SPR1 ,SPR0 (clock setting)=00 for 1Mbit/sec (fast enough)
0124                          *************************************
0125
0126 d088 86 5c                      LDAA #%01011100
0127 d08a b7 10 28                   STAA SPCR
0128
0129                          ************************************
0130                          *Send to FPGA
0131                           *       With SPI initialized we can
0132                           *       send the 3 bytes to FPGA
0133                          ************************************
0134 d08d  fe d1  00                 LDX $D100
0135 d090 bc d1  03            NXTB   CPX $D103
0136 d093 27 10                      BEQ FINISH
0137 d095 a6 00                      LDAA 0,X
0138 d097 b7 10  2a                  STAA SPDR  writes to  spi data
0139 d09a b6 10 29            WAIT   LDAA SPSR
0140 d09d 84  80                     ANDA #$80  this bit indicates SPI transfer of byte complete
0141 d09f 27  f9                     BEQ WAIT    wait until byte is complete
0142 d0a1 08                         INX
0143 d0a2 7e d0 90                   JMP NXTB
0144 d0a5 7e d0 00            FINISH JMP POLL
0145
0146
```

2. Verilog Code

```
//Author: Michael James Messina/Richard Trinh
//Date: December 3, 2000

module main(clk,reset,r,c,rowchoose,displayrow,func,leds,segs,SCK,MOSI,GETPAT);

        input clk,reset;
        input [3:0] r;
        input SCK,MOSI; //from the HC11
        output [2:0] c;
        output [2:0] rowchoose;
        output [8:0] displayrow;
        output [3:0] func;
        output [7:0] leds;
        output [6:0] segs;
        output GETPAT;
        wire slowclk,blink,resetflag,submit,correct,win,patternreceived;

        wire [8:0] anspattern,correctanspattern;
        wire [20:0] pattern;
        wire [3:0] key;
        wire [3:0] score;
        //assign pattern=21'b101100011110101000001; //test pattern
        //assign patternreceived=1;

        FSM fsm(clk,reset,patternreceived,submit,correct,score,resetflag,blink,blinkdone,leds,win,GETPAT);
        clkdiv cd(clk,reset,slowclk);
        keypad kp(slowclk,reset,r,c,key);
        display disp(slowclk,reset,pattern,anspattern,rowchoose,displayrow,func,win);
        determinecorrectpattern  dcp(clk,reset,pattern,correctanspattern);
        anspatterntoggle  apt(slowclk,reset,key,correctanspattern,blink,anspattern,correct,submit,blinkdone);
        segdisp sd(clk,reset,score,segs);
        shiftreg sr(SCK, reset,  resetflag, MOSI, pattern,  patternreceived);

endmodule

//////////////////////////////////////////////////////////////////////////////////////////////////////

module FSM(clk,reset,patternreceived,submit,correct,score,resetflag,blink,blinkdone,countout,win,GETPAT);
        //The following is the "standard" FSM setup
        //S0:get pattern:no display; ends when pattern done
        //S1:allow user to input answer (countdown): display;
        //   ends with user-submit or countdown done;  inc/dec score
        //S2: flash correct pattern; ends after 5 seconds
        //repeat
        input clk,reset;
        input submit,patternreceived,blinkdone,correct;
        output resetflag,blink;
        output win;
        output [7:0] countout; //output of countdown to  LEDs
        output [3:0] score; //keeps the score for the game
        output GETPAT;
        reg [1:0] state,nextstate;
        wire resetflag,blink,win;
        reg [20:0] pattern;
        wire gocount,countdone; //go count makes countdown go,  countdone tells FSM to take  anspattern as is
        wire submitorcount;     //high when either user presses submit or count is done => answer is submitted
        parameter S0=2'b00;
        parameter S1=2'b01;
        parameter S2=2'b10;
        parameter S3=2'b11;
        reg [3:0] score;           //for holding score to send to output
        reg inc;                   //holds increment command for  sk below
        scorekeep sk(clk,reset,inc,score);                   //module for keeping score for game (see next module)
        countdown cd(clk,reset,gocount,score,countout);
        assign win=(score==4'b1010);
        assign gocount=(state==S1);
        assign countdone=(countout==0);
        assign submitorcount=( countdone | submit);   //the pattern will be taken if submit is pushed or count done
        assign resetflag=((state==S1)&  patternreceived);    //reset the shift regi ster module (to reset  patrec)
        assign blink=(state==S2);                                          //blink if in state 2
        assign GETPAT=(state==S0);
//State Register
        always@(posedge clk or posedge reset)
                if (reset) state<=S2;
                else     state<=nextstate;

//State Logic
        always@(state  or  submitorcount or  blinkdone or correct or win or  patternreceived)
```

```
        case (state)
                S0:     begin
                        inc<=0; //we don't want to increment in S0
                        if (patternreceived)      nextstate<=S1;
                        else nextstate<=S0;     //if the pattern has completed transfer  goto S1
                                                 //otherwise stay in S0
                        end

                S1:     begin
                        if (submitorcount)     //if submit (B) has been pushed or count is done
                                begin
                                        inc<=(correct);  //send inc signal to  sk module to
                                                         //increment score
                                        nextstate<=S2;   //then  goto S2
                        end
                else
                    begin
                        inc<=0;                       //otherwise don't set inc
                        nextstate<=S1;                //and stay in S1
                    end
                end

                S2:     begin
                        inc<=0;                     //we don't want to increment in S2 either
                        if (blinkdone)
                                begin
                                        if(win) nextstate<=S3;    //if score=10 win!
                                        else    nextstate<=S0;    //else go to next pattern
                                end
                        else                  nextstate<=S2;        //otherwise stay in S 2
                end

                S3:     begin
                        inc<=0;
                        nextstate<=S3;
                        end
                default:
                        begin
                        nextstate<=S0;                      //by default, stay in S0
                        inc<=0;                             //again, no incrementing in S0
                        end
        endcase
endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module counter ( clk,reset,count);
        input clk,reset;
        output [20:0] count;
        reg [20:0]count;
        always@(posedge clk or posedge reset)
                if (reset) count<=0;
                else count<=count+1;

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module scorekeep(clk,reset,inc,score); //this holds the score
        input clk,reset,inc;
        output [3:0] score;
        reg [3:0] score;
        always@(posedge clk or posedge reset)
                if (reset) score<=0;
                else if (inc) score<=score+1;
                else score<=score;
endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module countdown( clk,reset,gocount,score,countout);

input  clk,reset,gocount;
input  [3:0] score; //level is represented by score (0 through 9)
output [7:0] countout;

reg [5:0] count;
reg [19:0] timer;  //instead of using a  clkdiv and dividing more, we will just use another divider
wire slowclk;      //this will give us a pulse once a second
wire [5:0] countout;
wire [5:0] value;
```

```
wire asynchreset;

assign slowclk=timer[19];                          //one 'count' each second (approx)
assign value={(4'b1110-score[3:0]),2'b00};         //each level will have 4 seconds less than the previous
                                                   //for 1st level time=60 seconds for 10 level time=20 seconds)
assign countout={2'b00,(value-count)};             //e.g level=0111 =>value=011100 (28 seconds for level 7)
=>output=28-count
assign asynchreset=~gocount;

        always@(posedge clk or posedge reset)
                if (reset) timer<=0;
                else if (~gocount) timer<=0;
                else timer<=timer+1;

        always@(posedge slowclk or posedge reset or posedge asynchreset)
                if (reset) count<=0;
                else if (asynchreset) count<=0;
                else count<=count+1;

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module keypad (clk,reset,r,c,key); //taken from lab4 solutions with minor changes

        input reset,clk;
        input [3:0] r;    //input rows from  keypd
        output [2:0] c;   //output columns to keypad
        output [3:0] key; //output of pressed key (for later decoding)

        reg [2:0] c;
        reg [3:0] key;
        reg gotkey;
        always@(posedge clk or posedge reset)

                if (reset)
                        begin
                        c<=3'b011;    //start column polling at col0
                        key<=4'b0000; //no key pressed
                        gotkey<=0;
                        end
                else if (&r)           //if all rows high, no key is being pressed
                        begin
                        gotkey<=0;
                        c<={c[0],c[2:1]}; //shift the columns to the right
                        key<=key;
                        end
                else if (~gotkey)
                        begin
                        c<=c;
                        gotkey<=1;
                        case ({r,c})
                                7'b0111_011: key<=4'b0001;
                                7'b1011_011: key<=4'b0100;
                                7'b1101_011: key<=4'b0111;
                                7'b1110_011: key<=4'b 1010;//A=Next pattern
                                7'b0111_101: key<=4'b0010;
                                7'b1011_101: key<=4'b0101;
                                7'b1101_101: key<=4'b1000;
                                7'b0111_110: key<=4'b0011;
                                7'b1011_110: key<=4'b0110;
                                7'b1101_110: key<=4'b1001;
                                7'b1110_110: key<=4'b1011; //B=Submit answer
                                default:  key<=4'b0000;
                        endcase
                        end
                else if (gotkey)
                        begin
                        key<=0;   //we only want the key to be taken once
                        end
endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module anspatterntoggle(clk,reset,key,correctanspattern,blink,anspattern,correct,submit,blinkdone);

        input clk,reset;
        input [3:0] key;
        input blink;
        input [8:0] correctanspattern;
        output [8:0] anspattern;
```

```
        output submit,blinkdone,correct;

        reg [8:0] anspattern;
        reg [8:0] correctansspattern;
        wire submit,correct;

        reg blinkdone;              //==1 when user wants next pattern
        reg [7:0] timer;            //this is used for the blinking state to be on 1/2 sec off 1/2 sec
                                    //we are using a slow clock (1/4096)  so we want to divide by 128 more
                                    //to get about 0.5 sec each
        reg on;                     //holds which part we are in for the blinking state

        assign submit=(key==4'b1011);
        assign correct=( correctanspattern==anspattern);

        always@( posedge clk or posedge reset)
                if (reset)
                        begin
                        anspattern<=9'b000000000; //all  LEDs off
                        timer<=0;
                        on<=1;   //we want the blinking state to start off (better looking)
                        blinkdone<=0;
                        end
                else if (~blink)
                        begin //this implies the  the user is inputting  leds to toggle
                                timer<=0;
                                on<=0;
                                blinkdone<=0;
                                case(key)
                                4'b0000: anspattern<= anspattern;
                                4'b0001: anspattern[8]<=anspattern[8]^1'b1;          //when keys 1-9 are
                                4'b0010: anspattern[7]<=anspattern[7]^1'b1;          //pressed the corresponding
                                4'b0011: anspattern[6]<=anspattern[6]^1'b1;          //bit will be toggled
                                4'b0100: anspattern[5]<=anspattern[5]^1'b1;          // by XORing with 1
                                4'b0101: anspattern[4]<=anspattern[4]^1'b1;
                                4'b0110: anspattern[3]<=anspattern[3]^1'b1;
                                4'b0111: anspattern[2]<=anspattern[2]^1'b1;
                                4'b1000: anspattern[1]<=anspattern[1]^1'b1;
                                4'b1001: anspattern[0]<=anspattern[0]^1'b1;
                                default: anspattern<= anspattern;                    //all other keys will leave
                                                                                     //the pattern unchanged
                                endcase
                        end
                else if (blink)                           //the correct pattern will be displayed blinking
                        begin
                                if (key==4'b1010)
                                        begin
                                                anspattern<=0;
                                                timer<=0;
                                                on<=0;
                                                blinkdone<=1;
                                        end
                                else //Next pattern button has not been pressed
                                        begin
                                                blinkdone<=0;
                                                timer<=timer+1;
                                                if (timer[7]==1)
                                                        begin
                                                                timer<=0;
                                                                if (on==1)
                                                                        begin
                                                                                on<=0;
                                                                                anspattern<=0;
                                                                        end
                                                                else //on=0
                                                                        begin
                                                                              anspattern<= correctanspattern;
                                                                              on<=1;
                                                                        end
                                                        end
                                                else
                                                        begin
                                                                anspattern<= anspattern;
                                                                on<=on;
                                                        end
                                        end
                        end
endmodule

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
module display( clk,reset,pattern,anspattern,rowchoose,ledrow,func,win);

        input clk,reset,win;
        input [20:0] pattern; //this is the pattern from the HC11
        input [8:0]  anspattern;
        output [2:0] rowchoose; //this will choose the current output row
        output [8:0] ledrow;
        output [3:0] func;

        reg [2:0] rowchoose;
        reg [8:0] ledrow;
        reg [3:0] func;

        always@ (posedge clk or posedge reset)

                if (reset)
                        begin
                        rowchoose<=3'b110; //start with one of the rows on
                                           //(the zero, these transistors NOT the base value)
                        ledrow<=0;         //clear the  ledrow
                        func<=0;           //clear the function
                        end
                else
                        begin
                        rowchoose<={ rowchoose[0],rowchoose[2:1]};
                        if (~win)
                                case (rowchoose)
                                        3'b110: ledrow<=~{ pattern[20:18],pattern[11:9], anspattern[8:6]};
                                                //the patterns are 9 bits long
                                        3'b011: ledrow<=~{ pattern[17:15],pattern[8:6], anspattern[5:3]};
                                                //but only 3 go in each row
                                        3'b101: ledrow<=~{ pattern[14:12],pattern[5:3], anspattern[2:0]};
                                        default: ledrow<=9'b111111111;
                                endcase
                        else //win! pattern= pluses
                                case (rowchoose)
                                        3'b110: ledrow<=~(9'b010010010); //the patterns are 9 bits long
                                        3'b011: ledrow<=~(9'b111111111);  //but only 3 go in each row
                                        3'b101: ledrow<=~(9'b010010010);
                                        default: ledrow<=9'b111111111;
                                endcase
                        case (pattern[2:0])
                                3'b000: func<=~(4'b0100);          //OR            OR and NOR are duplicated
                                3'b001: func<=~(4'b0001);          //XOR
                                3'b010: func<=~(4'b0010); //AND
                                3'b011: func<=~(4'b0100); //OR
                                3'b100: func<=~(4'b1001); //XNOR
                                3'b101: func<=~(4'b1010); //NAND
                                3'b110: func<=~(4'b1100); //NOR
                                3'b111: func<=~(4'b1100); //NOR
                                default: func<=~(4'b0000); //should never happen
                        endcase
                        end

endmodule

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

module clkdiv(clk,reset,slowclk); //taken from lab4 solutions

        input clk,reset;
        output slowclk;

        reg [11:0] count;

        always@(posedge clk or posedge reset)
                if (reset) count=0;
                else count=count+1;

        assign slowclk=count[11];

endmodule


/////////////////////////////////////////////////////////////////////////////////////////////////////////////

module determinecorrectpattern( clk,reset,pattern,correctpattern);
//This module takes the pattern (from HC11) of 2 X 9 bit patterns and 3 bit function code
//and determines the correct answer
```

```
        input clk,reset;
        input [20:0] pattern;
        output [8:0] correctpattern;      //the correct answer pattern

        wire [8:0] pat1,pat2;              //will hold the two  patterns
        wire [2:0] func;                   //will hold the function encoding
        reg [8:0] correctpattern;          //will hold correct answer pattern

        assign pat1=pattern[20:12];        //divide up pattern into individual patterns
        assign pat2=pattern[11:3];         // and the function to perform
        assign func=pattern[2:0];

        always@(posedge clk or posedge reset)
                if (reset)
                        correctpattern<=0;
                else
                        case (func)
                                3'b000: correctpattern<= (pat1 | pat2);//OR
                                3'b001: correctpattern<= (pat1 ^ pat2);//XOR
                                3'b010: correctpattern<= (pat1 & pat2);//AND
                                3'b011: correctpattern<= (pat1 | pat2);//OR
                                3'b100: correctpattern<=~(pat1 ^ pat2);//XNOR
                                3'b101: correctpattern<=~(pat1 & pat2);//NAND
                                3'b110: correctpattern<=~(pat1 | pat2);//NOR
                                3'b111: correctpattern<=~(pat1 | pat2);//NOR
                                default: correctpattern<=0;
                        endcase

endmodule

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module shiftreg (clk, reset, resetflag, datain, pattern, patternreceived);

input clk, reset, resetflag, datain;
output [20:0] pattern;                          // Holds the 21-bit pattern from HC11
output patternreceived;                         // Set when the 21 bits have been received from HC11

wire patternreceived;
reg [20:0] pattern;
reg [4:0] count;

assign patternreceived=((count==24)& ~resetflag);    //pattern is finished when 21st bit received
                                                     //patrec means we are ready to use pattern
        always@(posedge clk or posedge reset)
                if (reset)
                        begin
                                count<=0;
                                pattern<=0;
                        end
                else
                        begin
                                pattern <= {pattern[19:0], datain};
                                if (count==24) count<=0;  //reset count when HC11 is done (3 bytes worth)
                                else count<=count + 1;
                        end
endmodule

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module segdisp (clk,reset,score,segs);

input clk,reset;
input [3:0] score;
output [6:0] segs;

reg [6:0] segs;

        parameter ZERO= 7'b 0000001;
        parameter ONE=  7'b 1001111;
        parameter TWO=  7'b 0010010;
        parameter THREE=7'b 0000110;
        parameter FOUR= 7'b 1001100;
        parameter FIVE= 7'b 0100100;
        parameter SIX=  7'b 0100000;
        parameter SEVEN=7'b 0001111;
        parameter EIGHT=7'b 0000000;
        parameter NINE= 7'b 0000100;
        parameter A=    7'b 0001000;
        parameter B=    7'b 1100000;
        parameter C=    7'b 0110001;
```

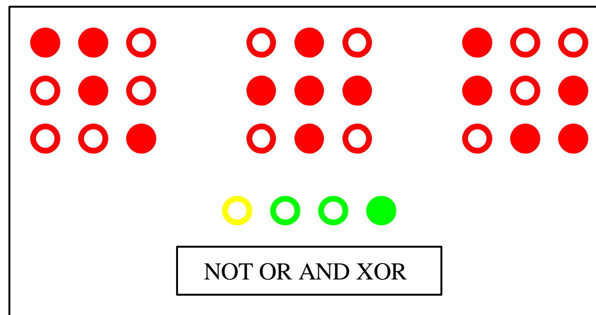
```
        parameter D=    7'b 1000010;
        parameter E=    7'b 0110000;
        parameter F=    7'b 0111000;

        always@(posedge clk or posedge reset)
                if (reset) segs<=7'b1111111;
                else
                        case(score)
                                0:  segs<=ZERO;
                                1:  segs<=ONE;
                                2:  segs<=TWO;
                                3:  segs<=THREE;
                                4:  segs<=FOUR;
                                5:  segs<=FIVE;
                                6:  segs<=SIX;
                                7:  segs<=SEVEN;
                                8:  segs<=EIGHT;
                                9:  segs<=NINE;
                                10: segs<=A;
                                11: segs<=B;
                                12: segs<=C;
                                13: segs<=D;
                                14: segs<=E;
                                15: segs<=F;
                        endcase
endmodule
```

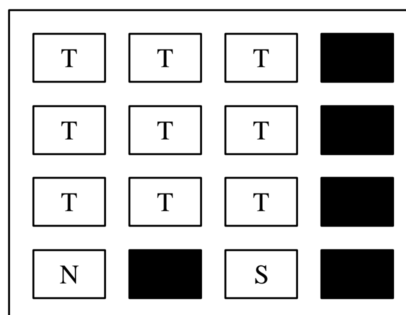
3. Flyer on How to Play

Digital Logic Game

by Michael James Messina and Richard Trinh
December 2000



To play the Digital Logic Game, first press the Reset button. When you're ready to play, press the N key on the keypad. You will see two patterns appear in the red LEDs and a digital logic function represented by the yellow and green LEDs. There are six possible logic functions: OR, AND, XOR, NOR, NAND, and XNOR. The goal of the game is to use the T keys to toggle the LEDs in the right-most array of red LEDs to match the correct pattern for the two displayed patterns and logic function. When you think you have the right answer, submit it with the S key before the binary countdown reaches zero. If you're right, you earn 1 point. Then press the N key again when you're ready for the next pattern. You win when you get 10 points. But be careful, the game gets harder as you go along!



Keypad Layout