컴퓨터 응용 설계 및 실험 (H/W)
제 15 주차 텀 프로젝트 최종 보고서
PIC16C57 마이크로프로세서 및 NFC 기반 도어락 장비 개발

제출일	2012, 12, 18	**전공**	정보컴퓨터공학부	
과목	컴퓨터 응용 설계 및 실험(HW)	**학번**	200724487	200824566
담당교수	양 세양 교수님	**이름**	양 희철	곽 두환

요약

본 텀 프로젝트에서는 PIC16C57에 기반 한 도어락 장비를 개발한다. 이 장비는 비밀번호의 입력을 통한 문을 여닫는 기능, 비밀번호 변경 기능 두 가지를 제공한다.

비밀번호의 입력은 비밀번호가 저장된 NFC 태그를 리더에 인식시키는 방식으로 이루어진다. 비밀번호의 변경은 키패드의 입력을 통해 이루어진다.

이 장비는 크게 FPGA 개발보드, AVR 개발보드, NFC 리더 세가지로 구성되어진다. FPGA개발보드에는 PIC16C57 마이크로프로세서와 함께 키패드, 모터 제어 모듈이 Verilog HDL로 구현되어 있다. AVR 개발보드는 ATMega128 마이크로프로세서가 장착되어 있으며, NFC 리더와의 UART 통신 및 FPGA 개발보드와의 통신을 담당한다. NFC 리더는 비밀번호가 입력된 NFC 태그를 읽어 AVR 개발보드로 전달하는 역할을 담당한다.

프로젝트 설명

1. 프로젝트 구성

1. 전체 구성

Figure 1은 개발한 도어락 장비의 전체 구성을 보여준다.

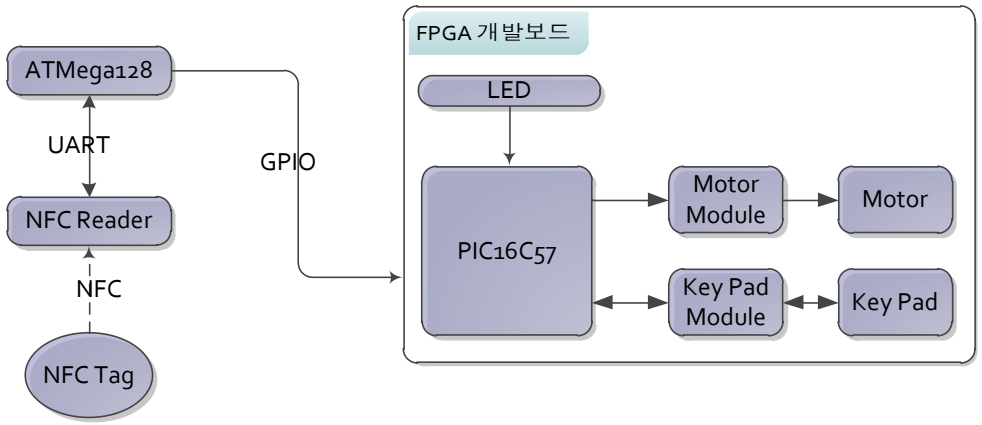

Figure 1 도어락 시스템 구조

2. PIC16C57

도어락 프로그램을 수행하는 중심 마이크로프로세서이다. 외부에서 들어오는 입력을 통해 문을 열거나 비밀번호를 수정하는 기능을 담당한다. 마이크로프로세서는 Verilog HDL로 구현하였고, 도어락 프로그램은 C로 작성하여 PIC의 Program Memory에 함께 Synthesis하였다.

3. 키패드 / 모터 Module

키패드는 변경 할 비밀번호를 입력할 때 사용한다. 키패드 모듈은 이 과정에서 키패드의 입력을 스캔 한 후 PIC로 전달한다.

모터는 올바른 비밀번호가 입력되었을 경우 동작한다. 모터 제어 모듈은 모터가 돌아 갈 수 있게 동작 신호를 차례로 주는 역할을 한다.

키패드 / 모터 모듈은 Verilog HDL로 작성되었다. 각 모듈은 PIC의 GPIO에 연결되어 있고, PIC의 도어락 프로그램이 이 모듈들을 제어한다.

4. NFC Reader – ATMega128 – FPGA

NFC 리더는 NFC 태그를 읽어내는 역할을 담당한다. 또한 본 텀 프로젝트에서 사용한 NFC 리더는 UART를 통해 외부와 통신을 한다. 하지만 PIC의 경우 UART 기능이 존재하지 않는다.

때문에 ATMega128은 NFC 리더와 FPGA를 연결하는 역할을 담당한다. NFC 리더와 ATMega128은 UART통해 연결되어서 NFC 리더가 읽은 태그 정보를 메모리에 저장한다. 또한 ATMega128은 PIC와 GPIO를 통해 연결되어 있다. 이를 통해 이 모듈은 읽은 NFC 태그에 저장된 비밀번호를 추출 한 후 PIC로 전달한다.

5. 시스템 동작

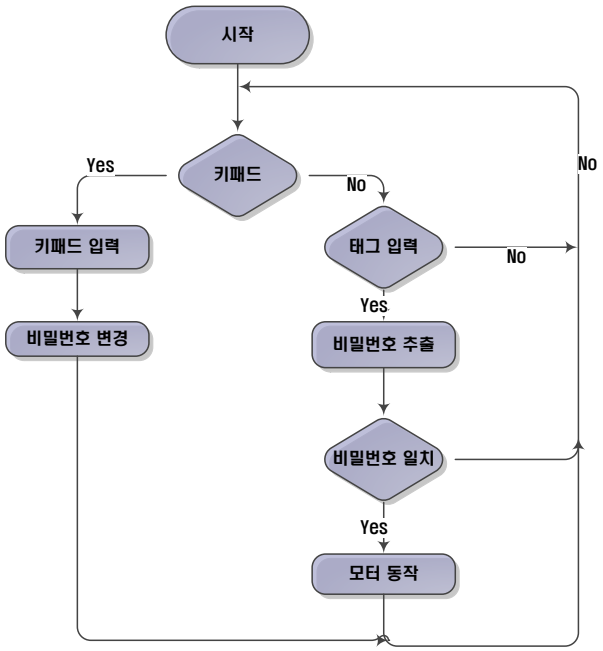

Figure 2 시스템 동작 흐름

Figure 2는 도어락 시스템의 전체적인 동작 흐름을 보여준다. 먼저 키패드 입력이 있을 경우 시스템은 입력 받은 4자리 번호를 비밀번호로 변경한다.

키패드 입력이 없이 태그를 입력이 들어왔을 경우 시스템은 태그에서 비밀번호를 추출 한 후 저장된 비밀번호와 일치한다면 모터를 동작시키고, 일치하지 않으면 모터를 멈추거나 동작시키지 않는다.

2. 설계 및 구현

1. PIC16C57

① Data Path

Figure 3은 구현한 PIC16C57의 Data Path의 Block Diagram이다.[1]

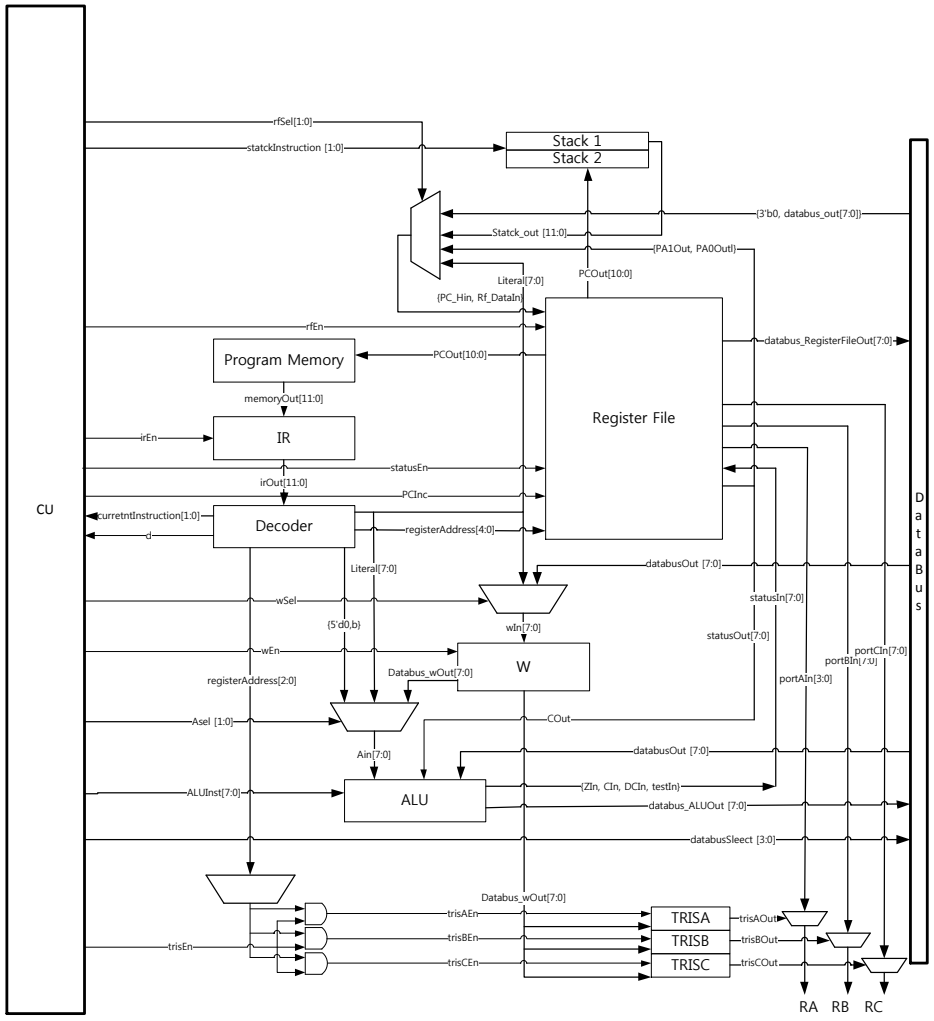

Figure 3 DataPath Block diagram

Table 1은 Data Path에 대한 요약이다.

모듈	설명
Data Bus	ALU / Register File / Status / W 의 입력에 대한 Mux
Stack	Function Call / Return 을 위한 Stack
Program Memory	수행 할 Instruction들을 저장하고 있는 ROM
Instruction Register	현재 수행 할 명령을 담고 있는 Register
Decoder	Instruction의 명령에 대한 Decoder
W Register	ALU 연산 결과를 저장하는 Register
ALU	각종 연산을 담당
TRIS A/B/C	IO Port의 I/O 여부를 설정하는 Register

[1] 실제 Data Path와 Control Unit의 Diagram은 다른 문서로 함께 첨부하였다.

Register File	Data Memory 및 Special Register들을 담고 있는 메모리

Table 1 각 Data Path 모듈 요약

Figure 4는 Data Bus에 대한 Verilog Code이다. 여러 모듈에서 들어오는 입력을 Multiplexing 하는 역할을 담당한다.

```
module DataBus(out, statusIn, ALUIn, registerFileIn, WIn,select, clk);
   output   reg   [7:0] out;
   input          [7:0] statusIn, ALUIn, registerFileIn,WIn;
   input          [3:0] select;
   input                clk;

   parameter W          = 4'd0;
   parameter ALU          = 4'd1;
   parameter registerFile  = 4'd2;
   parameter status       = 4'd3;


   reg      [7:0] nextData;
   always@(posedge clk)
   begin
      out <= nextData;
   end

   always@(select or ALUIn or registerFileIn or statusIn or WIn)
   begin
      case(select)
         ALU:          nextData = ALUIn;
         registerFile:  nextData = registerFileIn;
         status:       nextData = statusIn;
         W:            nextData = WIn;
         default:      nextData = 8'd0;
      endcase
   end
endmodule
```

Figure 4 Data Bus Verilog Code

Figure 5는 Stack에 대한 Verilog Code이다. Stack은 Level 1, Level 2 두 레지스터가 연결되어 있는 형식으로 구현되어 있다. Stack은 Push / Pop 명령을 입력으로 받는데, 각 명령에 따라 Level1, Level2 레지스터들의 데이터 이동이 일어난다.

```
case(instruction)
   PUSH:
   begin
      level1In = in;
      level2In = level1Out;

      level1En = 1'b1;
      level2En = 1'b1;
   end
   POP:
   begin
      level1In = level2Out;

      level1En = 1'b1;
      level2En = 1'b0;
   end
   default:
   begin

   end
```

Figure 5 Stack Verilog Code(일부)

Decoder는 Figure 6과 같이 IR로부터 입력 받은 명령을 해석하여 Control Unit으로 전달하는 역할을 담당한다. 추가로, 각 명령으로부터 Address 등의 여러 Field들도 함께 추출하여 Control Unit이나 Register File 등의 다른 모듈로 전달한다.

```
parameter MOVLW = 6'd28;
parameter TRIS = 6'd29;
parameter OPTION = 6'd30;

always@(*)
begin
   literal = instruction[8:0];
   d = instruction[5];
   BIT = instruction[7:5];
   address = instruction[4:0];
   casex(instruction[11:6])
      6'b000111:
      begin
         operation = ADDWF;
      end

      6'b000101:
      begin
         operation = ANDWF;
      end

      6'b000001:
      begin
         if(instruction[5] == 1)
         begin
            operation = CLRF;
         end
         else
         begin
            operation = CLRW;
         end
      end
```

Figure 6 Decoder (일부)

ALU 는 Control Unit으로부터 명령을 입력 받아 그에 대한 연산을 수행한다. 또한 연산 결과에 따라 Status Register에 입력할 값을 출력한다. Figure 7은 ALU의 구현이다.

```
parameter BSF = 8'd16;
parameter BTest = 8'd17;

always@(*)
begin
   Cout = 1'b0;
   temp = 4'b0000;
   test = (B[A[2:0]] == 1'b0);
   out = 8'd0;
   DC = 1'd0;
   case(command)
      AplusB:
      begin
         {Cout,out} = A + B;
         temp = A[3:0] + B[3:0];
         DC = temp[4];
      end
      AminusB:
      begin
         out = A - B;
      end
```

Figure 7 ALU 구현(일부)

Register File은 Data Memory 및 Special Register의 집합이다. 기본적으로는 Figure 8과 같이 입력한 Address에 대한 Read / Write 작업을 수행한다.

```
always@(posedge clk)
begin
   currentState <= memoryArray[address];
   if(rst)
   begin
      for(index = 0 ; index < 128 ; index = index + 1)
      begin
         if(index != PCL_ADDR) memoryArray[index] <= 8'd0;
      end
      memoryArray[PCL_ADDR] <= 8'd0;
      PCHOut <= 3'd0;
   end
   else
   begin
      if(write)
      begin
         memoryArray[address] <= dataIn;
```

Figure 8 Register File – Write 작업

Register File은 Figure 9와 같이 Special Register에 대한 작업도 함께 담당한다.

```
      case(address)
         PCL_ADDR:
            PCHOut <= PCHIn;
         STATUS_ADDR:
         begin
            for(index = 0 ; index < 8 ; index = index + 1)
            begin
               if(currentState[index] != dataIn[index])
               begin
                  currentState[index] <= dataIn[index];
               end
            end
            memoryArray[address] <= currentState;
         end
      endcase
   end
   else
   begin
      if(PCInc)
      begin
         {PCHOut, memoryArray[PCL_ADDR]} <= {PCHOut, memoryArray[PCL_ADDR]} + 11'd1;
      end
      else if(StatusEn)
      begin
         memoryArray[STATUS_ADDR] <= statusIn;
      end
   end
end
case(address)
   PORTA_ADDR:
      out <= {4'b0000,portAIn};
   PORTB_ADDR:
      out <= portBIn;
   PORTC_ADDR:
      out <= portCIn;
   default:
      out <= memoryArray[address];
endcase
```

Figure 9 Register File - Special Register 작업

마지막으로 Register File은 Figure 10과 같이 Indirect Addressing에 대한 처리도 수행한다.

```
always@(FSROut[6:5] or addressIn or memoryArray[FSR_ADDR])
begin
   address = {FSROut[6:5],addressIn};
   if(addressIn == 5'b00000)
   begin
      address = memoryArray[FSR_ADDR];
   end
   if(FSROut[6:5] == 2'b01 || FSROut[6:5] == 2'b10 || FSROut[6:5] == 2'b11)
   begin
      if(5'h00 <= addressIn && addressIn <= 5'h0F)
      begin
         address = {2'b00,addressIn};
      end
   end
end
```

Figure 10 Register File - Indirect Addressing

② Control Unit

Figure 11은 구현한 PIC16C57의 Control Unit 의 ASM Chart이다.

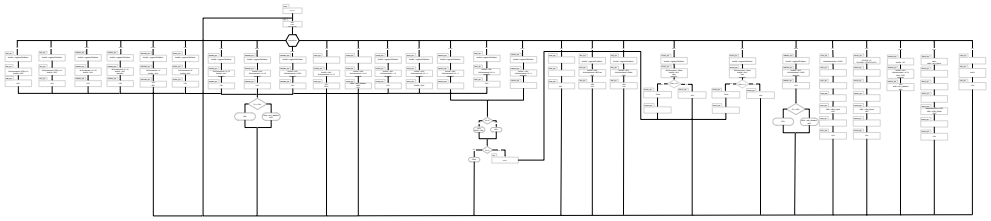

Figure 11 Control Unit ASM Chart

기본적으로 Control Unit은 Table 2와 같이 Q1 ~ Q4의 4 단계 Cycle을 가진다.

단계	역할	비고
Q1	Instruction Fetch / Decide	모든 명령 공통
Q2	Data Fetch	각 명령 마다 존재
Q3	Operation	
Q4	Data Save	

Table 2 Control Unit의 Cycle

Figure 12는 Control Unit이 Data Path에 전달할 신호들을 결정하는 부분이다.

```
BCF_Q2:
begin
   databusSelect = registerFile;
end

BCF_Q3:
begin
   ALUInst = BCF;
   databusSelect = ALU;
   ASel = 1'b1;
end

BCF_Q4:
begin
   rfEn = 1'b1;
   irEn = 1'b1;
end

BSF_Q2:
begin
   databusSelect = registerFile;
end

BSF_Q3:
begin
   ALUInst = BSF;
   databusSelect = ALU;
   ASel = 1'b1;
end

BSF_Q4:
begin
   rfEn = 1'b1;
   irEn = 1'b1;
end
```

Figure 12 Control Unit - Signal 생성(일부)

Figure 13은 Decoder로부터 입력받은 명령 정보를 통해 다음 State를 결정하는 부분이다.

```
Q1:
begin
   case(currentInstruction)
      DECODE_ADDWF: nextState = ADDWF_Q2;
      DECODE_ANDWF: nextState = ANDWF_Q2;
      DECODE_CLRF: nextState = CLRF_Q2;
      DECODE_CLRW: nextState = CLRW_Q2;
      DECODE_DECF: nextState = DECF_Q2;
      DECODE_DECFSZ: nextState = DECFSZ_Q2;
      DECODE_COMF: nextState = COMF_Q2;
      DECODE_INCF: nextState = INCF_Q2;
      DECODE_INCFSZ: nextState = INCFSZ_Q2;
      DECODE_IORWF: nextState = IORWF_Q2;
      DECODE_MOVF: nextState = MOVF_Q2;
      DECODE_MOVWF: nextState = MOVWF_Q2;
      DECODE_RLF: nextState = RLF_Q2;
      DECODE_RRF: nextState = RRF_Q2;
      DECODE_SUBWF: nextState = SUBWF_Q2;
      DECODE_SWAPF: nextState = SWAPF_Q2;
      DECODE_XORWF: nextState = XORWF_Q2;
      DECODE_BCF: nextState = BCF_Q2;
      DECODE_BSF: nextState = BSF_Q2;
      DECODE_BTFSC: nextState = BTFSC_Q2;
      DECODE_BTFSS: nextState = BTFSS_Q2;
      DECODE_ANDLW: nextState = ANDLW_Q2;
      DECODE_CALL: nextState = CALL_Q2;
      DECODE_RETLW: nextState = RETLW_Q2;
      DECODE_GOTO: nextState = GOTO_Q2;
      DECODE_IORLW: nextState = IORLW_Q2;
      DECODE_XORLW: nextState = XORLW_Q2;
      DECODE_MOVLW: nextState = MOVLW_Q2;
      DECODE_TRIS: nextState = TRIS_Q2;
      default: nextState = OTHERS_Q2;
   endcase
end
```

Figure 13 Control Unit - Next State 결정

2. 키패드 / 모터 Module

4 * 3 개의 키를 가지고 있는 키 패드는 각 열을 차례로 스캔하면서 눌러진 키를 알아내는 방식으로 동작한다. Table 3은 각 스캔별로 인식하게 되는 키를 나타낸다.

열	행	입력된 키
1	1000	1
	0100	4
	0010	7
	0001	*
2	1000	2
	0100	5
	0010	8
	0001	0
3	1000	3
	0100	6
	0010	9
	0001	#

Table 3 키패드 스캔 별 키 값

키패드 모듈은 Table 3에 따라 키패드를 스캔하면서 값을 인식하는 State Machine이다. Figure 14는 키패드 모듈에서의 State 이동을 수행하는 Verilog 코드이다.

```
always@(currentState or rst or scanData)
begin
   if(rst)
   begin
      nextState = IDLE;
   end
   else
   begin
      case(currentState)
         IDLE:nextState = COL0;
         COL0:nextState = COL1;
         COL1:nextState = COL2;
         COL2:nextState = COL0;
         default:nextState = IDLE;
      endcase
   end
end
```

Figure 14 키패드 모듈의 State

Figure 15는 각 상태 별로 입력된 키를 판별하는 Verilog Code이다.

```
COL0:
begin
   columnSel = 3'b001;
   case(scanData)
      4'b0001: keyData = 4'd1;
      4'b0010: keyData = 4'd4;
      4'b0100: keyData = 4'd7;
      4'b1000: keyData = 4'd10;
      default: keyData = 4'hF;
   endcase
end
COL1:
begin
   columnSel = 3'b010;
   case(scanData)
      4'b0001: keyData = 4'd2;
      4'b0010: keyData = 4'd5;
      4'b0100: keyData = 4'd8;
      4'b1000: keyData = 4'd0;
      default: keyData = 4'hF;
   endcase
end
```

Figure 15 각 상태 별 키 인식(일부)

스텝 모터는 Table 4와 같이 각 상태별로 다른 값을 차례로 입력시키면 동작하게 된다.

입력	상태			
	State 1	State 2	State 3	State 4
A	1	0	0	0
B	0	1	0	0
/A	0	0	1	0
/B	0	0	0	1

Table 4 스텝 모터 입력

모터 모듈은 Table 4에 맞추어서 만든 State Machine이다. Figure 16은 모터 모듈의 State 전환을 담당하는 Verilog 코드이다.

```
case(currentState)
    IDLE: nextState = STATE1;
    STATE1: nextState = STATE2;
    STATE2: nextState = STATE3;
    STATE3: nextState = STATE4;
    STATE4: nextState = STATE1;
    default: nextState = IDLE;
endcase
```

Figure 16 모터 모듈의 State 변환

Figure 17은 각 State 별로 모터에 신호를 주는 Verilog 코드이다.

```
always@(currentState)
begin
case(currentState)
   IDLE: motorControl = 4'b0000;
   STATE1: motorControl = 4'b1000;
   STATE2: motorControl = 4'b0100;
   STATE3: motorControl = 4'b0010;
   STATE4: motorControl = 4'b0001;
   default: motorControl = 4'b0000;
endcase
end
```

Figure 17 State 별 모터 입력

3. NFC Reader – ATMega128

NFC 리더는 NFC 태그를 인식한 후 Tag ID만을 추출하는 Polling Mode와 제어 명령을 통해 동작을 제어할 수 있는 Command Mode 두 가지 방식으로 동작한다. 본 조에서는 Figure 18과 같이 NFC 리더를 제어하였다.

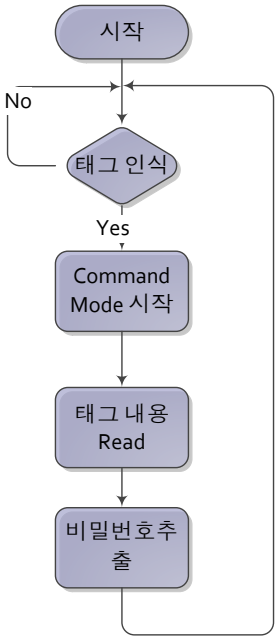

Figure 18 NFC 리더 제어

Table 5는 NFC 리더의 제어 명령이다. 이와 같은 명령들을 UART를 통해 리더

에 전달하면 그에 해당하는 동작을 실행 할 수 있게 된다.

Byte Format	비고
0xAA,0x00,0x01,0x00,0x01,0xFF	Command Mode 시작
0xAA,0x00,0x02,0x00,0x02,0xFF	Command Mode 종료
0xAA,0x05,0x01,0x00,0x04,0xFF	Tag ID Read
0xAA,0x05,0x02,0x02,0x30,0x07,0x32,0xFF	Tag Block Read

Table 5 NFC 리더 명령

Figure 19는 Table 5의 명령들을 NFC 리더로 전달하고 그 응답신호를 읽는 코드의 일부이다. 이 코드들은 내부적으로 UART를 통해 동작을 한다. 이 코드를 수행하고 난 후에는 메모리에 태그에서 추출 한 비밀번호가 저장된다.

```
//명령 모드 시작, ID Read
currentLength = memCat(buf,0,CommandStart,6);
currentLength = memCat(buf,currentLength,Ultra_ID_Read,6);
putNFCBytes(buf,currentLength);
getNFCBytes(debugBuf,20);

//Tag Block Read
currentLength = memCat(buf,0,Ultra_Read,8);
putNFCBytes(buf,currentLength);
getNFCBytes(debugBuf,23);

//비밀번호 추출
getPasswd(passwd, debugBuf);
```

Figure 19 NFC 리더 명령 코드(일부)

4. ATMeag128 – PIC16C57

```
PORTA = 0xFF;
PORTA = '*';
PORTA = 0xFF;
PORTA = passwd[0] & 0xFF;
PORTA = 0xFF;
PORTA = passwd[1] & 0xFF;
PORTA = 0xFF;
PORTA = passwd[2] & 0xFF;
PORTA = 0xFF;
PORTA = passwd[3] & 0xFF;
PORTA = 0xFF;
PORTA = '*';
PORTA = 0xFF;
```

Figure 20 비밀번호 전달

Figure 20은 추출한 비밀번호를 PIC로 전달하는 코드이다. FPGA와 ATMega128은 단순히 GPIO를 통해 연결되어 있기 때문에 GPIO Port에 데이터만 입력하면 비밀번호의 전달이 가능하다.

한편, 이 PIC에서 동작하는 도어락 프로그램은 '*'를 입력하면 비밀번호 입력 및 비교를 시작하도록 정의해 놓았다. 그렇기 때문에 비밀번호를 입력하기 전과 후에 '*'

문자를 넣어 주었다. 또한 각 입력 사이에는 다른 값이 들어간다는 신호로 0xFF를 주었다.

5. 결과물 실행 방법

Figure 21은 구현한 시스템의 전체 구성을 보여준다.

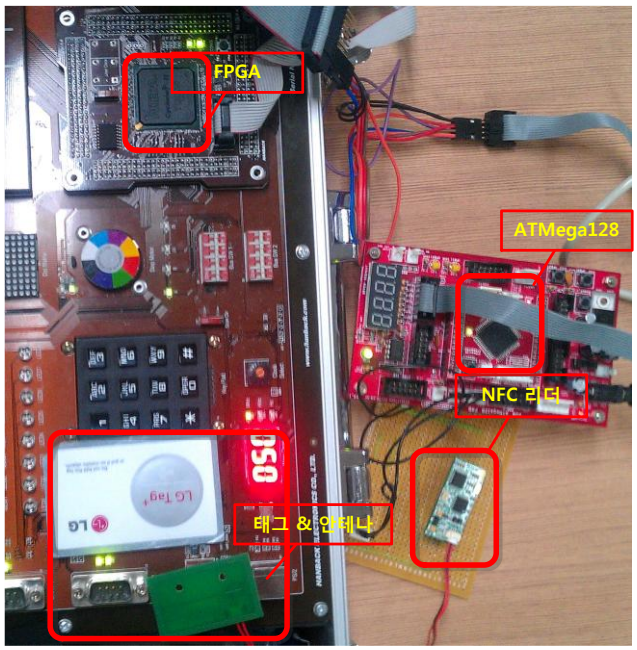

Figure 21 구현한 시스템 전체 구성

Figure 22는 안드로이드 스마트폰을 이용해서 NFC 태그에 비밀번호를 입력한 화면이다.

Figure 22 비밀번호를 입력 한 태그

Figure 21과 같이 구성한 후 안테나에 비밀번호가 입력된 태그를 인식시키면 비밀번호가 맞을 경우 Figure 23과 같이 LED가 켜짐과 함께 모터가 동작한다.

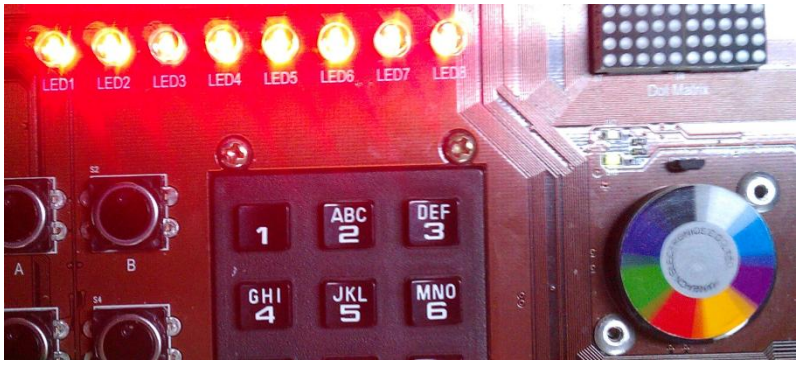

Figure 23 비밀번호를 제대로 입력하였을 경우

키패드는 비밀번호를 변경하기 위해 사용된다. '*" – 비밀번호 4자리 = '*' 순으로 입력하면 비밀번호가 변경되며, 이후에는 태그에 변경된 비밀번호를 저장하여야 한다.

결론

본 과목에서는 Verilog HDL을 이용해서 PIC16C57 마이크로프로세서를 설계 및 검증하는 것을 배웠다. 또한 구현한 PIC를 이용해 하나의 시스템을 직접 설계 및 구현하는 것을 텀 프로젝트로 수행하였다. 본 조에서는 간단한 도어락 시스템을 구현하였다.

텀 프로젝트를 수행하는 과정에서 지난 4년 동안 배운 전공 지식을 모두 사용 할 수 있었다. 전기회로 시간에 배웠던 지식을 바탕으로 시스템의 회로를 구성 하였고, C 언어를 이용해서 PIC 및 ATMega128에서 동작하는 Firmware를 구현하였다. 그리고 데이터 통신 및 컴퓨터 네트워크 시간에 배웠던 지식을 기반으로 NFC 및 UART 통신을 구현 할 수 있었으며 이 과정에서 두 시스템 간의 프로토콜 설계 경험도 할 수 있었다.

또한 논리설계, 시스템 소프트웨어, 컴퓨터 구조, 임베디드 시스템, VLSI 시스템 설계 등의 과목에서 배웠던 지식을 총망라 하여 PIC라는 상용 마이크로프로세서를 설계 할 수 있었다.

결국 이 과목의 텀 프로젝트는 회로 단계의 가장 밑바닥부터 실제 동작하는 응용 프로그램까지 전체 시스템을 모두 설계 한 결과이다. 비록 실제 결과물은 크지 않았지만 쉽게 접할 수 없는 경험을 할 수 있는 기회가 되었다고 생각한다.

첨부 문서

파일 / 디렉토리 명	내용
[컴퓨터종합설계][9조][15주차]텀프로젝트최종보고서.pdf	최종 보고서
Signal_table.pdf	Signal Table
CU_DP.pdf	Control Unit / Data Path Diagram
ATMega128.tar.bz	ATMega128 Source Code
PIC16C57.zip	PIC16C57 Source Code