HBE-COMBO II

User's Manual and Lab Guide

대전광역시 유성구 궁동 487-1
(042) 610-1111
www.hanback.co.kr

FG-CMMA-0904V10

❑ Revision History

Date	Version	Description	Revised by
2006-10-01	1st Edition(A)	Released initially	기술연구소

CONTENTS

그림 목차

01
HBE-COMBO II User's Manual & Lab Guide

개요

1. 개요

HBE-COMBO II는 현대의 전자, 정보통신 산업현장에서 필요로 하는 디지털 논리회로 설계에 대한 학교 교육에서 이론적인 교육환경을 벗어나 이론에서 얻은 결과를 직접 눈으로 확인 할 수 있도록 하는 환경을 제공하기 위한 디지털 논리 회로 설계 실습 장비입니다.

HBE-COMBO II는 산업 현장에서 디지털 회로설계에 많이 적용되고 있는 FPGA를 이용하여 사용자가 설계한 회로를 직접 하드웨어를 이용하여 동작시킴으로 이론교육에서 얻을 수 없었던 여러 가지 동작 현상을 실험을 통해 얻을 수 있으며 이와 아울러 학생들에게 디지털 논리 회로 설계에 대한 흥미를 유발 시킬 수 있습니다.

그리고 본 제품에서는 기존의 HBE-COMBO 제품의 특징인 두 회사의 모듈을 사용할 수 있도록 Altera사와 Xilinx사의 두 디바이스를 한 보드 내에서 사용할 수 있도록 메인 디바이스를 모듈화 하였습니다. 따라서 하나의 보드에서 메인 디바이스 모듈의 탈, 장착을 통해 두 회사에서 제공하는 디바이스를 경험해 볼 수 있도록 구성을 해 놓았습니다. 여기에 사용하는 디바이스도 두 회사에서 최근에 개발된 디바이스를 사용하도록 구성하고, 설계할 수 있는 용량도 대폭 늘려서 여유롭게 설계할 수 있는 구성을 가지고 있습니다.

사용자가 원하는 클럭을 손쉽게 입력 받아 사용할 수 있도록 클럭 컨트롤 블록을 구성하고 있고, 여기에서는 클럭 제어 스위치를 이용하여 16개의 클럭을 출력하게 구성하였습니다. 또한 디바이스 모듈에 별도의 사용자 클럭을 두어 원하는 오실레이터를 꽂아 사용할 수 있도록 구성하고 있습니다. 이 밖에도 동작에 필요한 설정을 단순화하여 사용상의 어려움을 최소화 하였으며 다양한 응용 모듈을 내장하여 별도의 장치 없이 충분한 실습이 이루어 지도록 하였습니다. 이외에도 확장 입출력 포트를 통해 제품에서 구성되지 않는 기능을 실습할 수 있도록 구성 하였습니다.

장비에 구성하고 있는 장치들을 살펴보면, VFD(Vacuum Fluorescent Display) 및 고휘도 LED를 두어 시인성을 더욱 강화하였습니다. 또한 다양한 입력 장치 (키 패드, 버튼 스

위치, 버스 스위치)를 사용하여 보드 내에서 직접 제어하여 실험할 수 있게 구성하고 있습니다. 또한 적외선 통신 실험을 할 수 있는 IrDA, USB to Serial 통신 실험을 할 수 있는 USB 포트, 자기센서를 이용한 스텝 모터 구동을 할 수 있도록 구성하였습니다. 또한 보드 내에 50핀 커넥터를 이용하여 다양한 확장 모듈을 장착하여 실험할 수 있도록 구성하고 있습니다.

HBE-COMBO II를 이용하면 디지털 논리회로 설계의 기본의 이론적인 부분을 벗어나 다양한 응용 모듈과 장치들을 이용한 설계에서 더 높은 회로 설계의 기술을 익히고 배우게 될 것입니다. 제품에 장착되어 있는 최신의 디바이스와 대용량의 디바이스로 설계에 대한 제약이 없이 설계를 할 수 있습니다. 또한 두 회사의 디바이스를 사용하여 설계하는 회사의 제약 없이 회로를 설계 할 수 있는 방법을 익히게 될 것입니다.

1.1 제품 특징

❑ Altera, Xilinx 디바이스의 모듈화

FPGA 디바이스의 유연성과 확장성을 고려하여 Altera와 Xilinx 디바이스의 교체 사용이 가능하도록 모듈화 하였으며 Altera사의 Cyclone II Series(3.5M ~ 7M System Gates)와 Xilinx사의 Spartan 3 Series(1M ~ 4M System Gates)를 사용할 수 있도록 구성하고 있습니다.

❑ 디바이스 모듈의 독립성

Device Module 내에 FPGA 디바이스와 Configuration 디바이스를 같이 두어 모듈의 Configuration Device에서 한번의 프로그램으로 다음의 전원 인가 시 Configuration ROM에 의한 데모 동작을 할수 있도록 구성하고 있습니다. 또한 별도의 오실레이터를 모듈에 장착하여 사용이 가능하도록 구성하고 있으므로 모듈에 전원 공급만으로 단독 사용이 가능하게 구성하고 있습니다.

❑ Clock Control Block

베이스 보드에 있는 50MHz의 오실레이터 출력을 Clock Control Block을 이용하여 0Hz ~ 50MHz의 16종류의 클럭으로 분주하여 사용할 수 있도록 구성하고 있습니다. 또한 디바이스 모듈에서 별도의 사용자 클럭이 있어 모듈에 사용하는 오실레이터 값이 그대로 전달되어 사용할 수 있도록 구성하였습니다. 따라서 베이스 보드와 디바이스 모듈의 2개의 클럭을 입력 받아 다양한 클럭 선택을 통한 설계가 가능하게 구성하였습니다. 이러한 두 종류의 클럭은 디바이스 별로 각각의 클럭 전용 핀에 연결되어 있어 제어 프로그램에서의 클럭 핀의 선택 따라 원하는 클럭을 선택하여 사용이 가능합니다. 이러한 클럭의 선택은 디바이스 모듈에 있는 User Clk EN 스위치를 이용하여 디바이스 모듈에 있는 클럭 입력을 제어할 수 있습니다.

❑ 장비의 확장성

사용자가 제작한 입출력 장치들을 인터페이스 할 수 있도록 50 pin x 2의 확장 포트를 지원하여 사용자가 보드 이외의 곳에 데이터를 출력하여 제어가 가능하도록 구성하고 있습니다. 또한 별도의 50핀 확장 커넥터를 통해 HBE-COMBO II용 모듈을 장착하여 다양한 실험을 하도록 구성하고 있습니다.

❑ 손쉬운 프로그래밍 작업

다양한 작업 (FPGA Download, PROM Program)에 사용되는 전용 케이블 모듈을 이용하여 PC와 장비를 연결하여 손쉽게 프로그램 할 수 있도록 구성하였습니다.

1.2 제품 구성

[그림 1-1] HBE-COMBOII 구성도

1.3 시스템 요구사항

1) PC 하드웨어 및 소프트웨어 사용 요건

❑ PC 하드웨어 :

- IBM 호환 펜티엄 II 이상
- 64MB 이상의 메모리
- ALTERA 프로그램 사용 시 (1GB 이상의 하드디스크 여유공간과 100MB 이상의 작업 공간)
- Xilinx 프로그램 사용 시 (2GB 이상의 하드디스크 여유공간과 100MB 이상의 작업 공간)
- Microsoft Windows 호환 Graphic Card
- Microsoft 호환 마우스
- 최소 1개 이상의 EPP/ECP 지원 프린트 포트 (LPT)
- CD-ROM 드라이브

❑ 소프트웨어 :

- OS : Windows NT 3.5 이상, Windows 2000, Windows XP
- 설계 프로그램 : Altera (Quartus II 5.0 이상), Xilinx (ISE Foundation Series 8.1i 이상)

2) FPGA 디바이스 규격

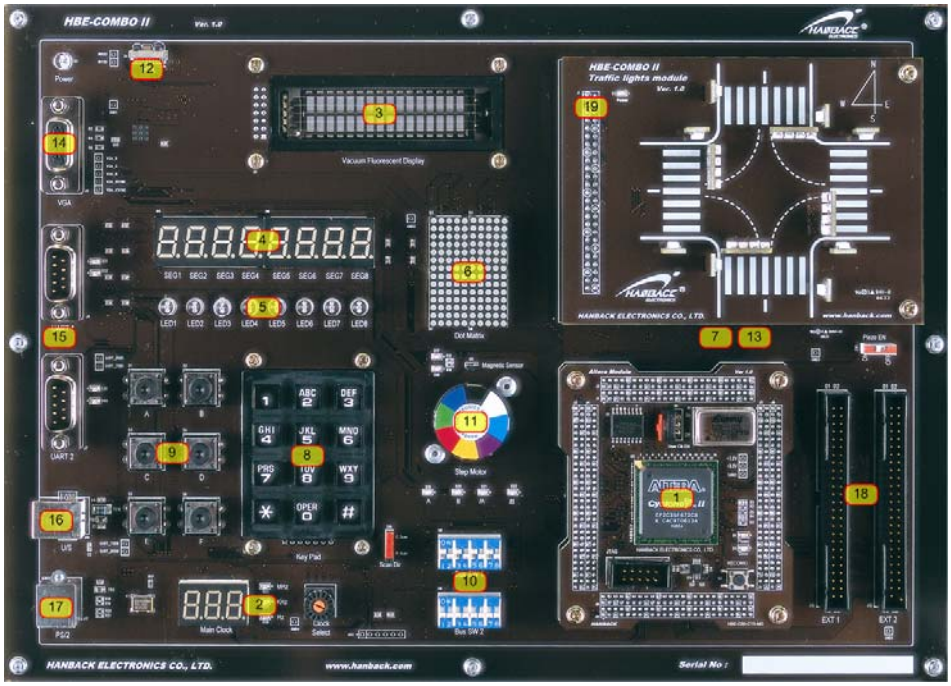
- ALTERA FPGA Device (3.5M ~ 7M System Gates)
- Cyclone II Series (EP2C35F672, EP2C50F672, EP2C70F672)
- Xilinx FPGA Device (1M ~ 4M System Gates)
- Spartan 3 Series (XC3S1000FG676, XC3S1500FG676, XC3S2000FG676, XC3S4000FG676)

3) Configuration PROM 규격

Altera사의 Serial Configuration Device와 Xilinx사의 Flash In-System Programmable Configuration PROMS의 각 회사의 전용 FPGA Configuration ROM을 사용합니다.

- Altera FPGA용 PROM : EPCS16SI16N
- Xilinx FPGA용 PROM : XCF08PVO48C 또는 XCF16PVO48C

1.4 보드 구성 및 각 부분 명칭



[그림 1-2] HBE-COMBO II 보드구성 및 각 부분 명칭

Altera사의 Cyclone II Series를 사용하는 경우에는 Quartus II를 사용하여 디바이스를 제어하고, Xilinx사의 Spartan-3 Series를 사용하는 경우에는 ISE를 이용하여 제어가 가능합니다. 이러한 2개의 소프트웨어는 각 회사의 홈페이지에서 웹 버전을 무료로 다운 받아서 사용할 수 있습니다.

1) FPGA 모듈

Altera 및 Xilinx FPGA 디바이스를 겸용으로 장착하여 사용 가능하도록 커넥터로 구성하였습니다. 따라서 사용자가 사용하려는 FPGA Module을 선택 장착하여 사용이 가능합니다. 여기에서 선택하는 모듈에 따라서 설계 소프트웨어도 달리 해 주어야 합니다. 이렇게 설계된 환경은 전용 다운로드 케이블을 이용하여 디바이스 모듈의 JTAG 커넥터로 FPGA 디바이스와 Configuration PROM에 다운할 수 있습니다.

따라서 FPGA 모듈은 전원 공급을 통해 디바이스에서 다운이 가능하게 구성하였습니다. 또한 FPGA 디바이스는 RAM 타입의 디바이스로 전원 OFF시 디바이스 내부의 데이터가 사라짐으로 다시 전원을 ON 하더라도 이전의 동작이 계속 유지가 되지 않습니

다. 그래서 전원을 켤 때마다 디바이스에 재 프로그램 해 주어야 합니다.

이런 불편한 점을 보완하기 위해 별도의 Configuration ROM을 두어, 한번의 프로그램 만으로 데이터가 사라지지 않고 RESET 버튼을 통해 Configuration ROM을 통해 미리 저장된 데이터를 넘겨주어 동작이 가능하게 구성을 하였습니다. 현재 FPGA 모듈 에서 사용하는 Configuration ROM은 Altera (EPCS Series), Xilinx (XCF Series)를 사용하고 있습니다.

그리고 베이스 보드로부터 클럭의 입력을 받아 사용하고, 모듈 내에 오실레이터를 장착하여 User Clock을 입력 받아 사용할 수 있도록 구성하였습니다.

2) 클럭 제어 블록

메인 클럭은 Base Board에서 제공하는 50MHz를 로터리 스위치를 이용하여 0Hz ~ 50MHz까지의 16개로 클럭 형태로 분주 하여 사용하도록 구성하고 있습니다. 또한 모듈에 별도의 User Clock을 달아서 사용할 수 있도록 구성을 해 놓았습니다. 사용자가 사용하려는 오실레이터의 값이 그대로 FPGA 디바이스에 전달되어 사용을 할 수 있도록 구성이 되어 있습니다.

따라서 사용자가 원하는 클럭을 오실레이터를 꽂으면 그 클럭이 바로 디바이스에 전달되어 사용이 가능합니다. 이 User Clock은 옆에 잉카 스위치를 두어서 사용하는 오실레이터가 없을 때 스위치를 OFF 시켜서 클럭 핀을 안정화 시켜 놓았습니다. 사용하는 클럭은 몇 개의 클럭 입력 전용 핀을 이용해서 입력을 받고 있으므로 메인 클럭과 User Clock을 동시에 받아서 회로를 설계 할 수 있습니다.

3) VFD

16문자 x 2라인을 출력할 수 있도록 구성된 VFD 모듈입니다. VFD란 진공 형광 표시 장치[Vacuum Fluorescent Display]로써 저속 전자선에 의한 형광체의 여기 발광 현상을 이용한 표시 장치 입니다. 따라서 HBE-COMBO의 Text LCD 보다 도트 자체적으로 발광을 하기 때문에 밝기 면에서 우수하여 시인성이 뛰어나고 응답 속도가 빠른 특징이 있습니다.

동작은 영문, 숫자, 특수문자 등을 문자표를 보고 데이터 값을 보내서 표시가 가능합니다. 또한 코드의 수정으로 문자의 밝기를 4가지 모드로 조절할 있는 특징이 있습니다.

4) 7-Segment Array

총 8자리의 숫자를 표시할 수 있도록 구성된 7-Segment 모듈 입니다. 각 Segment에서 사용하는 핀의 수를 줄이기 위해 Segment의 Common 핀을 이용하여 Segment를 하나씩

선택하고, 이렇게 선택된 Segment에 data를 주는 샘플링 방식을 이용하여 제어합니다. 따라서 FPGA I/O가 제한되어 있는 곳에서 적은 핀으로 보다 많은 7-Segment를 제어할 수 있습니다.

5) High light LED

총 8개의 딥 타입의 고휘도 LED로 구성되어 있습니다. 따라서 보다 선명한 빛을 통해 LED의 동작을 확인 할 수 있습니다.

6) Dot Matrix LED

10x14 도트로 구성된 Dot Matrix LED로서 최대 4자의 영문자 또는 1자의 한글을 표시할 수 있도록 구성되었습니다.

7) Piezo

특정 주기의 클럭 소스를 인가할 경우 멜로디를 출력하는 장치입니다. 10KHz ~ 1Hz의 클럭 소스를 사용하여 다양한 멜로디를 출력 할 수 있습니다. 그림에서는 신호등 모듈 밑에 위치하고 있습니다.

8) Key pad

3 X 4의 키 패드 모듈을 사용하여 숫자 1 ~ 0과 #,*를 표현 할 수 있는 스위치를 사용하고 있습니다. 스위치의 동작은 스캔 방식을 이용하여 제어를 하도록 구성이 되어 있습니다. 스캔으로 사용할 수 있는 라인인 Column과 Row 라인을 “Scan Dir" 스위치를 통해 선택 할 수 있습니다.

9) 버튼 스위치

A ~ F까지의 입력을 할 수 있도록 버튼 스위치 6개로 구성이 되어 있습니다. 이 버튼에 대한 표시는 보드에 표기되어 있습니다. 또한 스위치에는 채터링을 방지하기 위한 회로가 추가 되어 있어 버튼의 입력을 확실히 받을 수 있습니다..

10) 버스 스위치

16비트의 버스 입력을 위한 Dip 타입의 스위치입니다. 이 스위치에는 채터링 방지 회로가 내장되어 있지 않으므로 버튼 스위치의 용도로 사용할 경우 정상적인 입력이 되지 않을 수 있습니다.

11) Step Motor

4개의 데이터 라인을 통해 1상 여자 방식, 2상 여자 방식, 1-2상 여자 방식으로 모터를 구동하게 됩니다. 따라서 이러한 데이터의 신호에 따라서 모터가 가지고 있는 정해진 각도에 따라 모터 축이 회전하는 원리를 가지고 있는 모터를 구성해 놓았습니다. 또한

모터의 회전을 감지하기 위해 자기센서를 부착하고 있어, 모터의 제어를 더욱 다양하게 할 수 있습니다.

12) IrDA

적외선 주파수 스펙트럼 내의 모아진 광선을 이용하여 비교적 짧은 거리 내에 있는 송, 수신기 사이의 무선 데이터 통신에 이용할 수 있는 블록을 구성하고 있습니다. 송, 수신기를 하나의 모듈로 구성한 HSDL-3600 1개를 보드 가장 자리에 배치하고 있어 다른 장비와 연동하여 서로간의 적외선 통신 실습을 구현할 수 있습니다.

13) Memory

512K Bit의 SRAM과 16M Bit의 Flash Memory로 구성되어 있으며 SRAM과 Flash Memory는 각각 별도의 데이터 라인을 통해 독립적으로 제어가 가능합니다. 이러한 메모리들의 핀을 제어하여 메모를 영역을 제어할 수 있도록 구성해 놓았습니다. 그림에서는 신호등 모듈 밑에 위치하고 있습니다.

14) VGA 포트

컬러 VGA 모니터를 구동할 수 있도록 디지털 RGB를 아날로그 RGB 신호로 변환해 줄 수 있는 D/A Converter가 연결되어 최대 1024 X 768 해상도에 최대 24-bit RGB 컬러를 구현할 수 있도록 구성되어 있습니다.

15) RS-232 통신 포트

PC 등의 개발 호스트와 RS-232 포트와의 통신을 위한 시리얼 포트입니다. 9핀 Male로 2개가 장착되어 있습니다.

16) USB to Serial 포트

시리얼 통신을 하기 위한 포트로써 RS-232 포트를 지원하지 않는 호스트에서 USB 포트를 이용하여 실험할 수 있는 장치 입니다. 지원하는 디바이스로 FT232를 사용하고 있고, http://www.ftdichip.com/에서 윈도우용 드라이버를 설치하여 사용하면 됩니다.

17) PS/2 포트

PC의 입력 장치로 사용되는 PS/2 키보드나 마우스를 연결해 키보드 컨트롤러나 마우스 컨트롤러를 설계해 볼 수 있는 PS/2 커넥터 2개로 구성되어 있습니다.

18) 확장 포트

외부 Application 장치나 사용자께서 제작한 장치를 연결하기 위한 포트로 50핀 Male 커넥터로 구성되어 있으며 총 2 포트를 제공합니다.

19) 확장 포트(Daughter)

장비 내에 50핀 확장 커넥터에 확장 모듈을 장착하여 실험할 수 있는 포트입니다. 이 모듈에는 신호등, 자판기부터 오디오 코덱을 실험할 수 있는 모듈 등이 구성되어 있습니다.

1.5 제품 규격

구 분	사 양
FPGA Device	- Altera : Cyclone II Series (EP2C35~70F672), User I/O : 422 ~ 475 - Xilinx : Spartan-3 Series (XC3S1000~4000FG676), User I/O : 391 ~ 489
Configuration ROM	- Altera : EPCS16 - Xilinx : XCF08PVO48 or XCF16PVO48
SRAM	256K x 16 bit High Speed Static RAM
Flash	128 Mbit Embedded Flash Memory (Option)
Clock	50MHz base board Oscillator 1EA, Ext User clock
VGA	1024 x 768 Resolution 24-Bit True-color VGA port 1EA
USB	USB to Serial Interface
Serial	RS232 UART Port 2EA(FPGA I/O)
PS/2	PS/2 Keyboard or PS/2 Mouse port 1EA
Keypad	3 x 4 Key-Pad Switch
Input Button	User Push-button 6EA
Input Bus Switch	User 8-bit DIP Switch 2EA
LED	User LED Displays 8EA
7-Segment	4-Digit 2EA(Total 8-Digit)
VFD	16 x 2 Vacuum Fluorescent Display 1EA
Dot-Matrix	14 x 10 Dot-matrix 1EA(7 x 5 Dot-matrix 4EA)
Piezo	5V Input Piezo 1EA
Motor	Step Motor with Phase LED 1EA
Sensor	Magnetic Sensor 1EA
IrDA	Compliant 4MB/s 3V Infrared Transceiver
Ext. port	25 X 2 I/O Expansion port 2EA(Support total 92 pin), 5V Supply
Ext. port(Daughter)	25 X 2 I/O Expansion port 1EA(Support total 44 pin), 5V, 12V Supply
Power	Input 220V/60Hz AC/Output : +5V, +3.3V, +2.5V, +1.8V, +1.2V
CD	전자 매뉴얼 및 예제 CD 1EA
Manual	사용 설명서 & 실습 매뉴얼
Accessory	- 220V power cable : 1EA - Serial cross cable : 1EA - Parallel cable : 1EA - Altera ByteBlaster II download cable or Xilinx Parallel download cable : 1EA
Board Size	FPGA board : 96 X 122 (mm), base board : 312 X 221 (mm)

02
HBE-COMBO II User's Manual & Lab Guide

처음 사용하기

2. 처음 사용하기

본 장비는 다음의 네 부분으로 크게 구분할 수 있습니다. FPGA부, 클럭 제어 부, 내부 장치부, 확장 부로 나누어지며 장비를 사용할 때는 이들을 조작하여 필요한 동작을 위한 준비를 하여야 합니다.

FPGA부는 디바이스를 이용한 직접적인 제어를 하는 곳으로, HBE-COMBO II의 Base board의 커넥터를 통한 FPGA 디바이스를 장착하여 사용이 가능합니다. FPGA 모듈은 Altera, Xilinx 사의 2 종류의 디바이스를 모듈화 하고 있기 때문에, 모듈의 탈 장착으로 두 회사의 디바이스를 모두 사용할 수 있습니다.

따라서 HBE-COMBO II 장비에서는 벤더의 제한 없이 사용이 가능합니다. 각 디바이스 모듈의 구성은 FPGA 디바이스, 각 디바이스의 PROM, 전원, 리셋 단, JTAG port가 구성되어 있습니다. 따라서 커넥터를 통한 전원의 공급만으로 디바이스 모듈의 단독 구동이 가능합니다.

클럭 제어 부는 HBE-COMBO II의 base board에 구성되어 있습니다. 이 부분은 FPGA 디바이스에 공급하는 클럭을 제어하는 부분입니다. FPGA 디바이스는 오실레이터를 통해 클럭을 공급 받아서 사용하는 환경을 가지고 있습니다. 하지만 보드에 장착되어 있는 클럭은 50MHz의 주파수를 가지고 있는 것으로 사용자가 설계 환경에서 사용하려는 클럭은 이러한 주파수 값을 분주하여 사용해서 사용해야 하는 불편함이 있습니다. 따라서 이러한 불편을 해소하기 위해 제어부를 두어 클럭을 분주해서 FPGA 디바이스로 공급해 주는 역할을 하고 있습니다.

이러한 클럭의 공급은 0Hz에서 50MHz까지의 16분주를 Clock Select 스위치를 이용하여 FPGA 디바이스로 공급이 가능합니다. 이러한 클럭의 값은 클럭 제어부의 7-Segment에서 입력되는 주파수를 확인할 수 있습니다. 7-Segment에서 주파수의 숫자를 확인할 수 있고 7-Segment 옆에 있는 3개의 LED를 통해 MHz, KHz, Hz를 확인할 수 있습니다. 이러한 클럭은 Base board의 50MHz의 클럭을 분주하여 디바이스로 공급해준 값입니다.

이 외에 디바이스 모듈에서 별도의 오실레이터를 사용하여 주파수를 공급해 줄 수 있습니다. 이 때, Base board의 주파수는 0Hz로 해야 하고, 디바이스 모듈의 클럭 제어 스

위치를 이용하여 User Clock을 활성화 시켜 주어야 합니다. 이 상태에서 모듈에 Dip 타입의 오실레이터를 장착하여 디바이스에 주파수 공급이 가능합니다. 여기에서 사용하는 오실레이터는 Half, Full의 두 가지 타입이 모두 가능하고 사용하는 오실레이터의 값이 그대로 FPGA 디바이스로 전달 됩니다.

내부 장치부는 HBE-COMBO II 장비에 구성되어 있는 장치들로, 포트를 이용한 외부의 장비와 통신을 위한 부분과 내부의 장치를 제어하는 부분들이 있습니다. 세부적으로 살펴보면 입 출력 포트로는 먼저 VGA 포트로 모니터 제어를 위한 것과, 2개의 UART 포트를 이용하여 PC등과의 시리얼 통신 실험을 위한 부분이 있습니다. 또한 USB 포트를 이용한 시리얼 통신 실험을 할 수 있습니다. 마지막으로 PS/2 포트를 이용하여 키보드나 마우스를 이용한 제어 실험을 해 볼 수 있습니다.

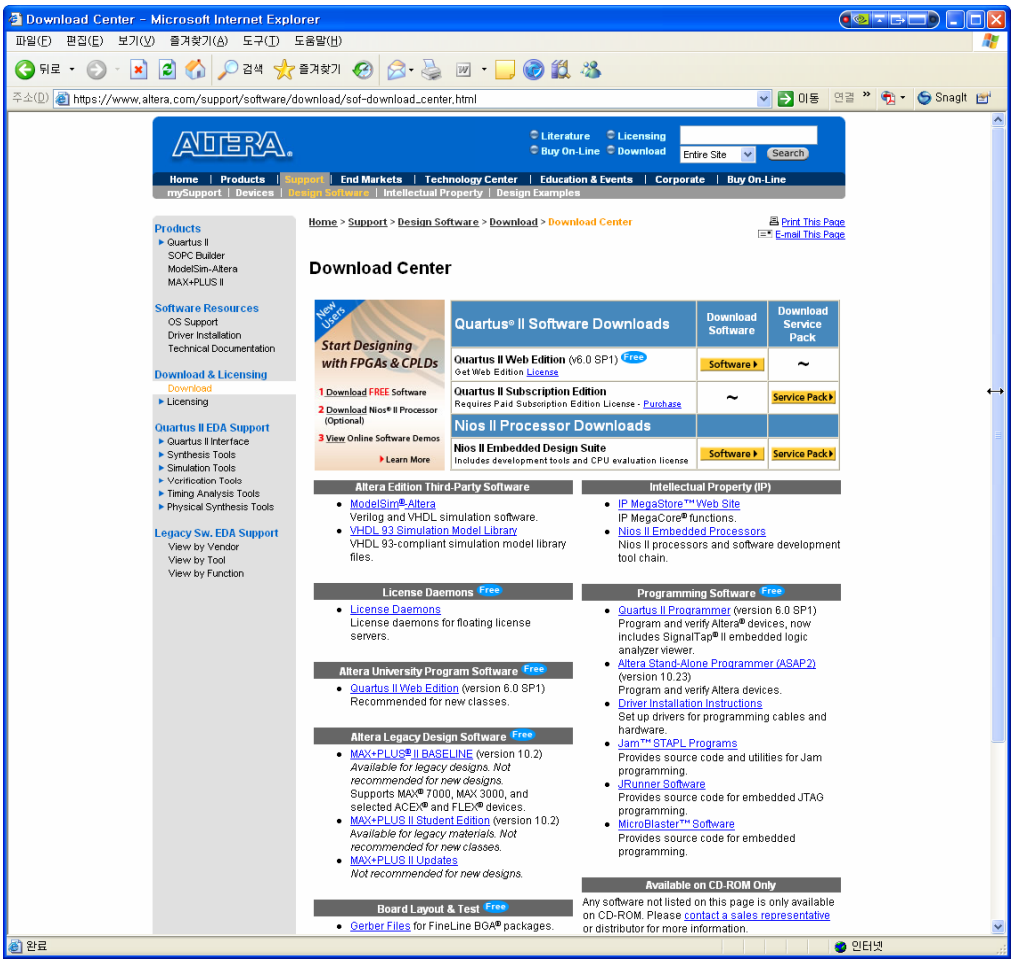
장비 내부에 있는 실습 장치들로는 Display 장치로 LED, 7-Senment, Dot Matrix등이 있습니다. 또한 VFD(Vacuum, Fluorescent Display)가 있습니다. VFD는 예전의 Text LCD와 제어와 동작은 비슷하지만, 각 dot별 형광 물질에 의한 발광을 하고 있으므로 밝기 면에서 우수한 성능을 보이고 있습니다. 입력 장치로는 키 패드, 버튼 스위치, 버스 스위치의 다양한 입력 스위치를 사용하여 입력으로 사용이 가능합니다. 또한 Piezo를 사용하여 주파수에 따른 음을 조절하여 멜로디 출력을 할 수 있는 부분이 있고, 스테핑 모터를 사용하여 모터 구동 실험을 할 수 있습니다. 마지막으로 적외선 센서를 이용한 통신 실험을 할 수 있는 IrDA 모듈을 사용하고 있습니다.

확장부는 50핀 확장 커넥터 3개를 사용하여 FPGA 디바이스를 디지털 신호를 장비 외부의 장치로 확장해서 사용할 수 있는 부분입니다. 디바이스 모듈 오른 쪽에 있는 EXT 1, EXT 2의 확장 커넥터는 5V의 전원을 포함한 확장이 가능합니다. 따라서 기존의 한백전자 FPGA Application Module 모두 제어 가능합니다. 이러한 모듈은 신호등, 퍼즐에서 엘리베이터까지 다양한 응용 모듈을 HBE-COMBO II에서 제어가 가능합니다.

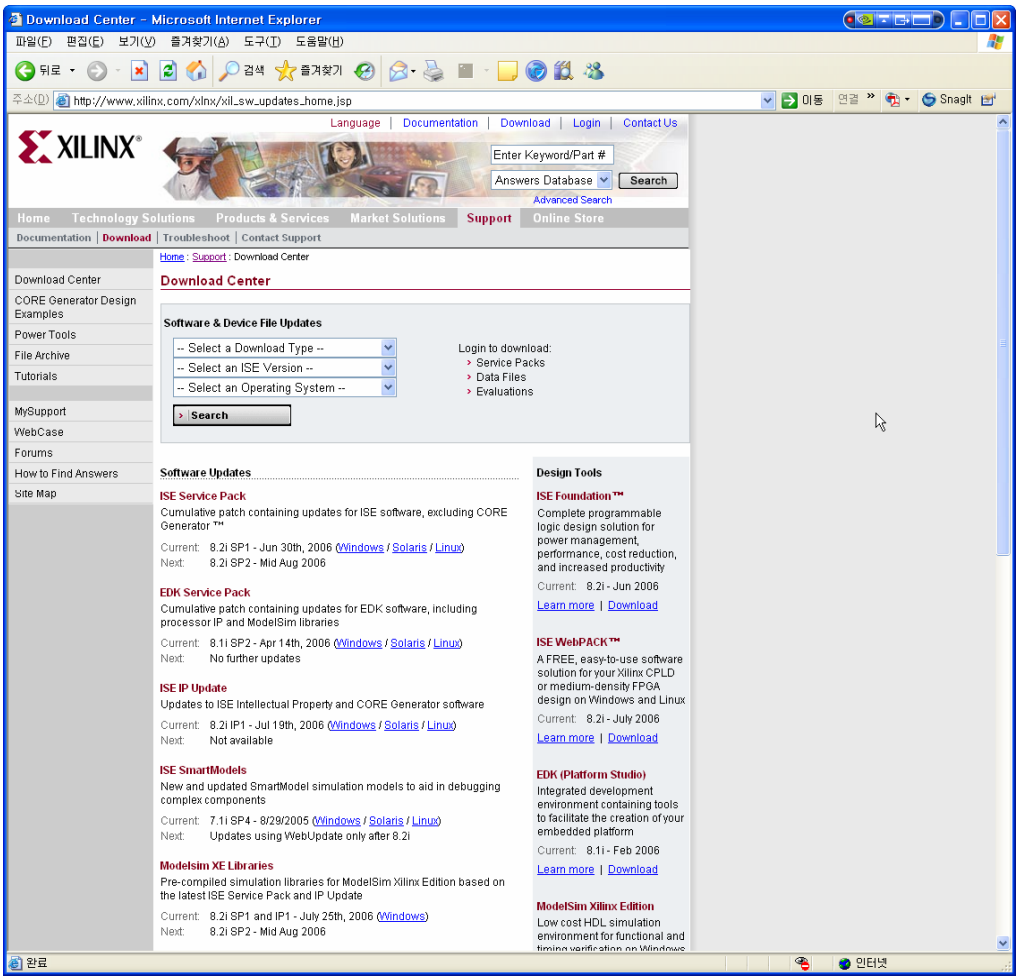
그리고 FPGA 디바이스 모듈의 위에 배치하고 있는 EXT 3은 50핀의 확장 커넥터로 5V, 12V의 전원을 공급 받아 사용이 가능합니다. 이 확장 커넥터는 HBE-COMBO II용 확장 모듈을 장착하여 사용할 수 있는 부분입니다. 이러한 보드의 종류로는 신호등, 자판기, ADDA, Stereo Audio Codec Board등이 있습니다. 따라서 이러한 모듈을 HBE-COMBO II 장비에 장착하여 다양한 실험을 해 볼 수 있습니다.

다음은 장비 사용을 위한 순서를 정리하였습니다.

- ❑ 본 장비를 활용하기 위해서는 Altera사 또는 Xilinx에서 제공하는 회로 설계 소프트웨어인 Quartus II, ISE가 준비 되어야 합니다. 이러한 설계 소프트웨어는 각 디바이스 벤더의 홈페이지에서 제공하는 무료 소프트웨어를 다운받아 설치하면 디바이스 제어가 가능합니다. 다음의 그림에서 각 벤더의 다운로드 창을 확인할 수 있습니다.



[그림 2-1] ALTERA Download Center



[그림 2-2] XILINX Download Center

❑ 사용할 FPGA 모듈을 선택하여 그림과 같이 장착합니다. 장착 시 주의할 사항은 커넥터가 정확히 일치하게 연결되지 않은 상태에서 전원을 인가할 경우 FPGA 디바이스가 손상될 위험이 있으므로 주의하여야 디바이스를 장착하여야 합니다. 초기에 제품에는 디바이스가 장착되어 있으므로 바로 전원을 인가하여도 됩니다.

[장착전]

[장착후]

❑ 제품에 포함되어 있는 전원 케이블을 제품의 오른 쪽 측면이 있는 전원 연결 포트에 연결합니다. 전원연결 포트에 스위치가 꺼져 있는지 확인하신 후 전원 케이블을 전원 플러그에 연결합니다. LPT용 케이블을 PC의 LPT (프린트) 포트에 연결하고 다른 한쪽은 각 디바이스 모듈에 맞는 다운로드 전용 케이블과 연결해 줍니다. 이러한 다운로드 케이블은 디바이스 모듈의 JTAG 포트와 연결합니다.

[전원케이블 연결]

[Download 케이블 연결]

❑ 보드 메인 전원 스위치를 on하여 Base 보드와 FPGA 모듈의 전원 LED에 불이 제대로 들어오는지 확인하고 전원 LED에 불이 들어오지 않을 경우 전원 케이블이 제대로 연결되었는지 또는 제품에서 쇼트 (단락)된 부분이 없는지 확인하기 바랍니다.

❑ 전원이 정상적으로 들어 올 경우 JTAG 커넥터를 통해 설계 소프트웨어의 프로그램 창을 이용하여 디바이스를 검색할 수 있습니다. Altera Quartus II에서는 Programmer에서 Auto Detect 버튼을 클릭하면 JTAG 커넥터와 연결된 Altera 디바이스 (EP2C35F672)가 보이게 됩니다. Xilinx ISE에서 iMPACT 에서 Initialize Chain으로 디바이스 검색을 하면, 연결된 디바이스 (XCF08, XC3S1000F676)가 화면상에 보이게 됩니다.

❑ 이렇게 디바이스 검색을 통한 연결 상태 및 전원을 확인을 해 보고 장비에 직접 다운로드 하여 미

리 설계된 설계 소스를 이용하여 장비의 동작을 확인해 볼 수 있습니다. 툴의 다운로드 창을 이용하여 장비에 있는 디바이스에 다운로드 합니다. 다운로드가 완료되면 장비가 사용자가 설계한 상태대로 동작을 하게 됩니다. 이 때 동작의 이상이 있거나 다운이 정상적으로 이루어 지지 않을 경우, 모듈의 전원이나 케이블의 연결 상태를 확인하고 다시 다운해 보기 바랍니다.

제품 동작은 다음과 같이 크게 3가지로 구분되며 각각의 동작 상태는 다음과 같습니다. 그리고 각각의 사용법은 사용하는 FPGA 디바이스에 따라 조금씩 다르므로 자세한 동작 방법은 다음 부분에 나와 있는 내용을 참고하기 바랍니다.

1) FPGA Download (Download)

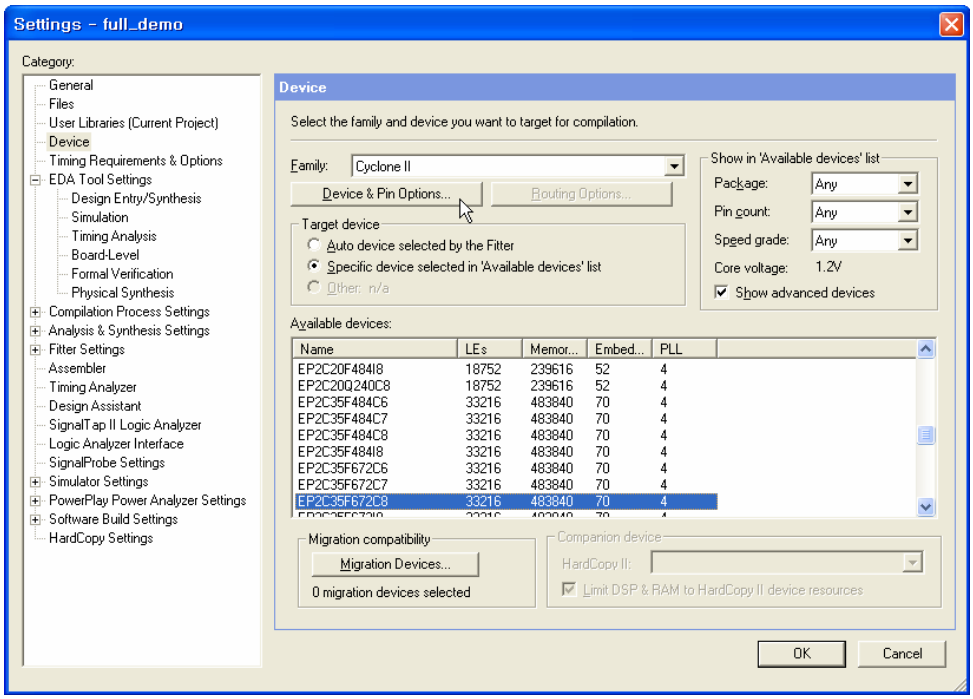
사용자가 설계한 회로를 FPGA 디바이스로 직접 프로그램 데이터를 전송하여 동작을 확인 하는 방법입니다. 다운로드에 의해 바로 동작을 확인할 수 있다는 장점은 있으나 제품의 전원을 off할 경우 다운로드 된 데이터가 디바이스에서 지워지기 때문에 다시 동작을 할 경우 다운로드를 해 주어야 합니다.

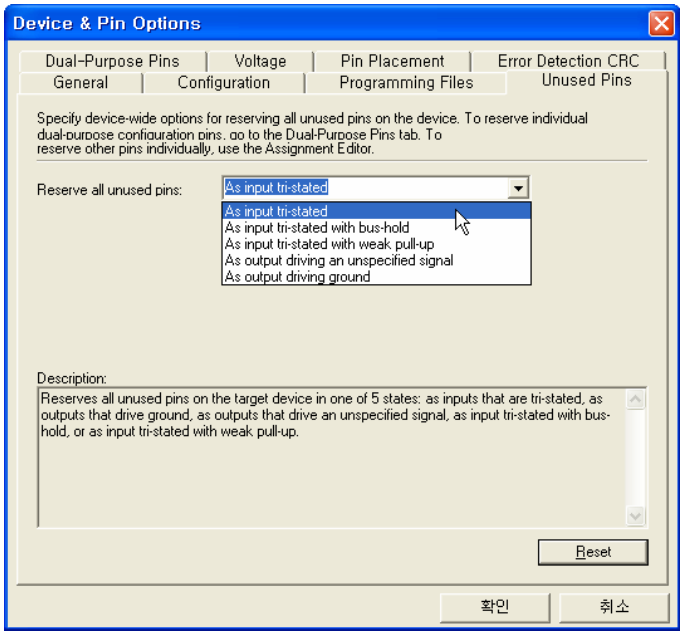
이 방법은 FPGA 디바이스가 SRAM 타입이기 때문에 전원에 따른 데이터의 손실을 보게 되는 것 입니다. 디바이스에 프로그램 하기 전에 주의할 사항은 컴파일 시, Setting 부분의 디바이스 옵션 창에서 디바이스의 사용하지 않는 핀에 대해 tri-stated buffer 상태로 설정을 해 두어야 합니다. 이렇게 함으로써 프로그램 후 디바이스에 대한 손상을 줄일 수 있습니다. 이러한 상태의 설정이 없다면 디바이스의 과열 반응의 원인이 됩니다.

이 작업은 다음에 나오는 그림과 같이 Quartus II 메뉴의 Assignments => Settings에서 이러한 작업을 설정해 줍니다. 여기서 활성화 되는 창에서 왼쪽 Category 창에서 Device를 선택하고, 나타나는 오른 쪽 창에서 Device & Pin Options.. 버튼을 클릭합니다. 다음 그림처럼 활성화 되는 창에서 위의 버튼 중에 Unused Pins를 선택해 줍니다. 그리고 Reserve all unused Pins를 체크하고 “확인” 버튼을 눌러 설정을 마칩니다.

이러한 설정을 하고 컴파일을 하면, 디바이스에 프로그램 시 발열 현상을 막을 수 있습니다. Xilinx 디바이스의 경우 별도의 설정 없이 프로그램 파일의 설정을 마치고 바로 다운해 주면 됩니다.

- ALTERA Device & Pin Setting





2) Configuration ROM Write (Write)

FPGA 디바이스는 앞서 설명한 것과 같이 SRAM으로 구성되어 있어 전원이 off되면 프로그램 데이터가 소거되어 전원을 on할 때마다 다시 프로그램을 다운로드 하여야 합니다. 이것을 개선하기 위해 Configuration PROM을 사용하여 FPGA 프로그램 데이터를 PROM에 저장하여 전원이 인가 될 때 자동으로 데이터를 FPGA로 다운로드 하여 장비가 동작하도록 합니다.

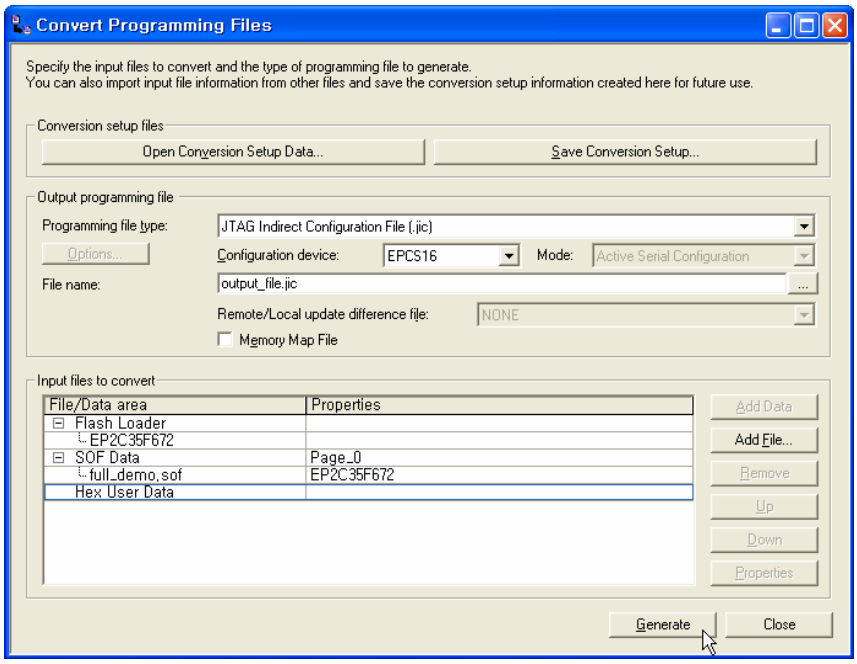
Configuration PROM을 프로그램 하는 방법은 Altera사와 Xilinx사의 tool의 메뉴에서 이러한 작업을 할 수 있습니다. 이러한 방법은 다음에 나오는 설명을 통해 두 회사의 디바이스에 대한 Configuration 방법을 참고로 해 보기 바랍니다.

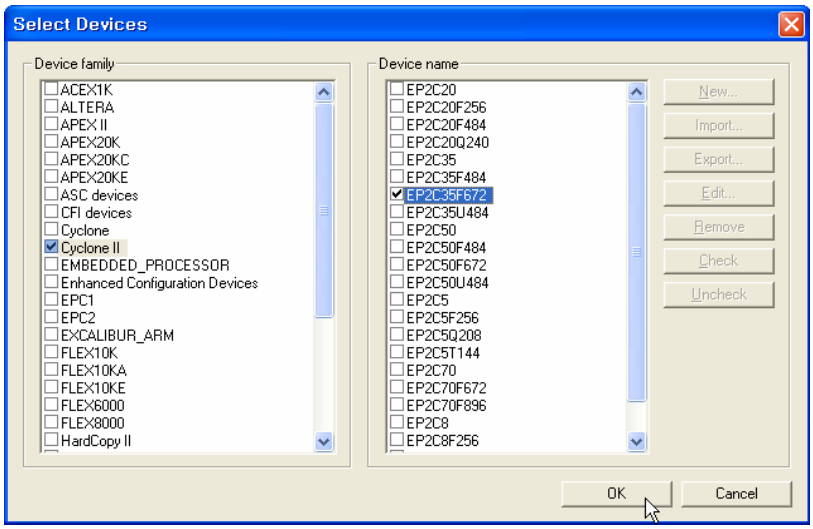
① Altera PROM Configuration

Altera 사에서 나오는 Serial PROM을 Write하기 위해서는 FPGA Device의 Serial Flash Loader를 이용하는 방법을 이용하고 있습니다. 따라서 FPGA Device에 있는 Serial Flash Loader에 먼저 프로그램 후 이러한 장치가 FPGA Device와 연결되어 있는 Serial Configuration Device에 직접 프로그램 하는 방식을 이용하고 있습니다.

따라서 tool에서 먼저 확장자가 .sof(Altera FPGA Device download file) 파일을 디바이스의 Serial Configuration Device에 프로그램 할 수 있는 파일인 .jic 파일로 변환을 해 주어야 합니다. 따라서 Quartus II의 메뉴에 있는 File -> Convert Programming File에서 이러한 파일 변환 작업을 할 수 있습니다. 참고로 여기서 실행하는 메뉴는 .sof 파일을 가지고 Altera 디바이스에 적용해 주기 위한 파일로 변환해 주는 작업을 하는 창이 됩니다. 다음의 창에서 이러한 파일의 변환 방법이 설명되어 있습니다.

- Convert Programming File





위의 그림을 통해 파일을 .sof => .jic 파일로 변환하는 방법을 보여주고 있습니다. 다음에서 이러한 설정이 어떻게 하는지 설명하도록 하겠습니다.

❑ Programming file type : JTAG Indirect Configuration File (.jic)

▶ 먼저 Programming file type을 변환하려는 Serial Configuration Device에 맞는 파일로 선택을 해 줍니다.

❑ Configuration device : EPCS16

▶ Write 하려는 Serial Configuration device를 선택해 줍니다. FPGA Module 디바이스 이름을 보고 확인할 수 있습니다.

❑ File name : 생성하려는 파일이름

▶ 사용자가 생성하려는 파일이름을 적어줍니다.(예 : full_demo.jic) 이때 확장자는 .jic 로 선택해 줍니다. 이때 저장하려는 폴더도 지정을 해서 파일 이름을 적어 주어야 합니다. 폴더를 지정하지 않으면 프로젝트가 선언된 root 폴더 (최상위 폴더)에 저장이 됩니다.

❑ Flash Loader : Cyclone II => EP2C35F672 ~ EP2C70F672

▶ Flash Loader는 현재 어떤 FPGA Device의 Loader를 사용할 것인지 선택해 주는 작업으로 현재 FPGA Module에 사용하고 있는 Cyclone II Family에 EP2C35F672를 선택해 줍니다. 이 작업은 오른 쪽에 나와 있는 그림에서 확인해 볼 수 있습니다.

❑ SOF Data : 현재 설계한 .sof 파일

▶ 여기에서는 변환하려는 대상 파일을 선택해 주어야 합니다. 현재 설계한 프로젝트의 FPGA Device 전용 프로그램 파일인 .sof 파일을 선택합니다.

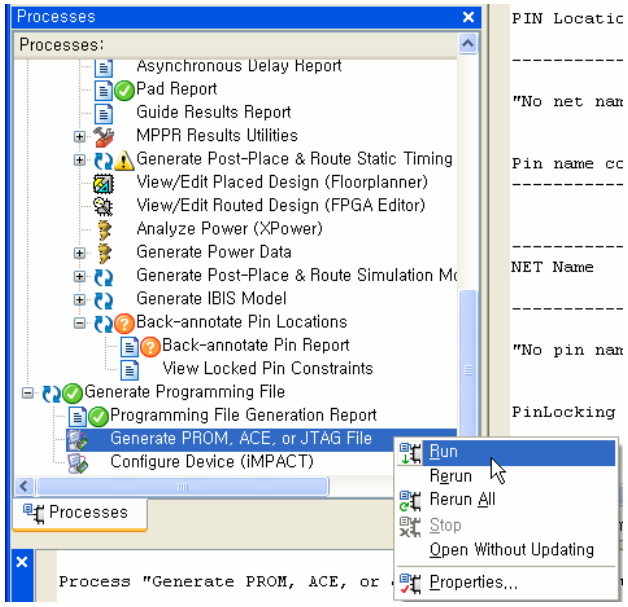
이렇게 설정을 마쳤으면 Generate 버튼을 클릭해서 파일을 생성하면 됩니다. 이렇게 생성된 파일을 가지고 Altera사의 Serial ROM에 프로그램 해 주면 됩니다.

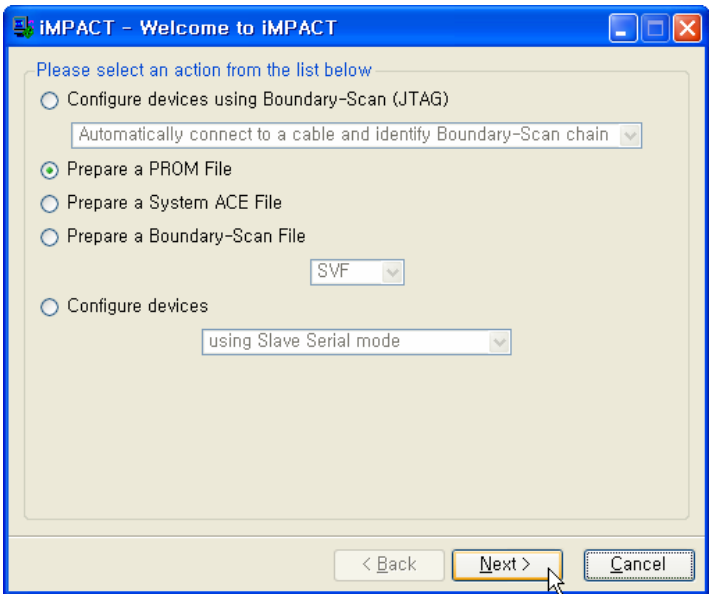
② Xilinx PROM Configuration

Xilinx 메인 디바이스도 Altera 디바이스와 같이 RAM 타입의 디바이스이기 때문에 별도의 ROM 타입의 디바이스를 같이 사용하여 이러한 점을 보완해 주고 있습니다. 여기에서 사용하는 PROM은 Xilinx사에서 나온 전용 Configuration PROM인 XCF08PVO48을 사용하고 있습니다.

따라서 여기도 마찬가지로 Xilinx사의 tool인 ISE를 가지고 파일의 변환 작업을 수행하여야 합니다. Spartan 3를 가지고 컴파일 된 최종 파일을 이용하여 변환 작업을 수행합니다. 다음의 그림을 통해 이러한 변환 작업을 수행하는 방법이 나와 있습니다.

- Generate PROM, ACE, or JTAG File





위의 그림에서는 .bit 파일을 Configuration PROM에 사용할 수 있는 파일로 변환을 위한 작업을 보여주고 있습니다. 따라서 ISE Processes 창에서 컴파일을 수행 후, 위의 그림과 같이 Generate PROM, ACE, or JTAG File를 실행 시킵니다.

이때 주의할 점이 Generate Programming File의 Properties에서 FPGA Start-Up Clock을

CCLK로 설정을 해 주어야 합니다. 이러한 작업은 Properties를 활성화 시킨 다음에 Category를 Startup Options을 선택합니다. 그리고 나타나는 하위 메뉴에서 FPGA Start-UP Clock을 CCLK로 설정하고 Generate Programming File을 실행 시킵니다.

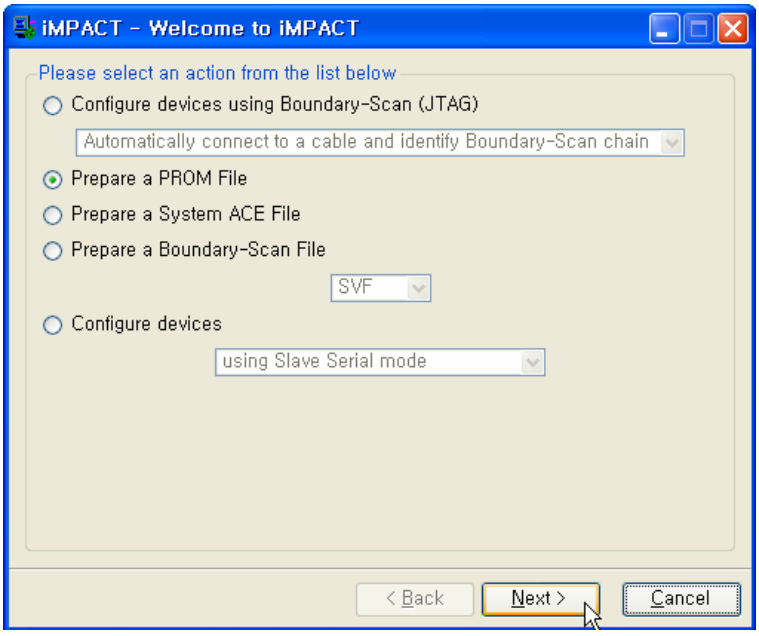
따라서 Generate PROM, ACE, or JTAG File을 실행하기 전에 이러한 작업이 완료 되어 있어야 합니다. 이러한 작업은 다운 방식에 따라 어떠한 다운 방식의 클럭을 이용하여 FPGA 디바이스에 프로그램 할 것인지 결정해 주는 작업으로 PROM으로 다운을 할 경우 이러한 작업이 완료가 되어 있어야지만 PROM에 의한 동작이 이루어 질 수 있습니다.

이렇게 설정을 하고 Generate PROM, ACE, or JTAG File을 실행 시키면 Generate PROM, ACE, or JTAG File 그림의 오른 쪽과 같은 창이 나타나고, PROM Configuration File에 대한 변환 작업을 시작하게 됩니다.

다음부터는 PROM 변환을 위한 작업 순서를 보여주고 있습니다.

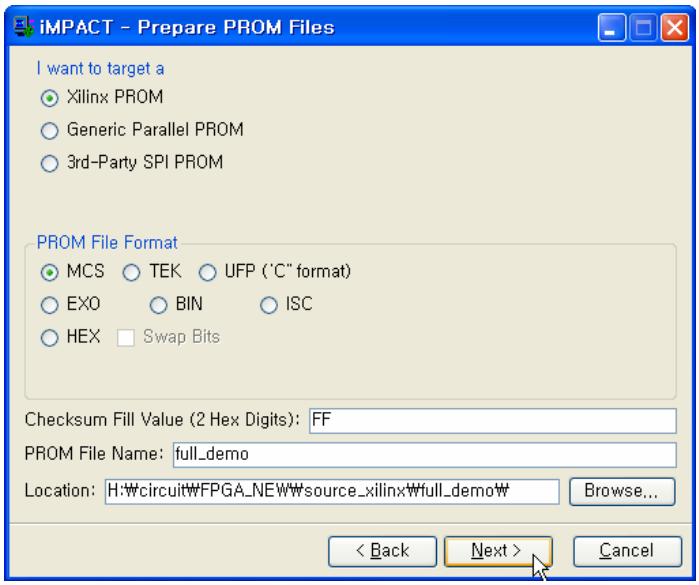
- **Welcome to iMPACT**

❑ Impact를 실행해 디바이스 관련 작업을 하는 설정 창입니다. 여기에서는 타겟 디바이스에 관한 프로그램 과정부터 현재 설정을 하려는 PROM 관련 파일 변환 작업을 수행할 수 있는 곳입니다. Xilinx PROM에 적용해 주기 위한 파일 변환을 위한 작업을 수행하려고 하고 있으므로 'Prepare a PROM File' 에 대한 부분을 체크하고 다음 설정 창으로 넘어갑니다.



- **Prepare PROM Files**

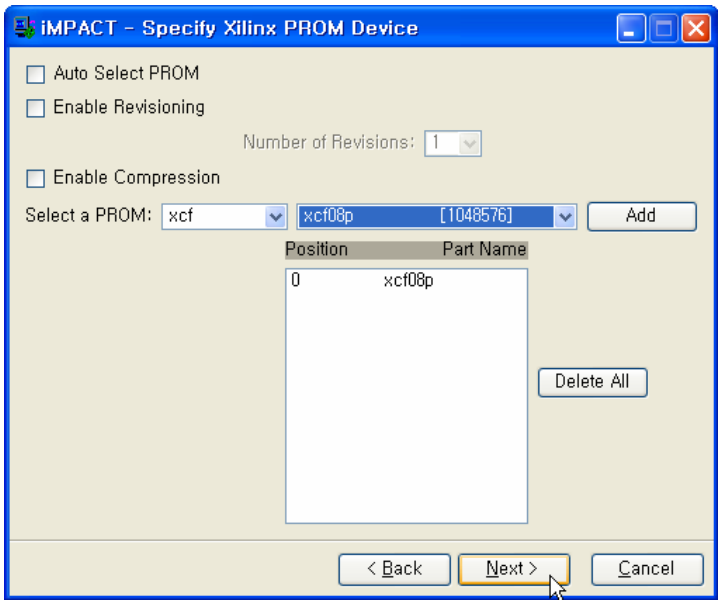
❑ 생성할 파일에 관한 설정을 해 주는 창입니다. 창에서 윗 부분에 잇는 “I want to targer a ~ “ 부분에서 어떠한 종류의 PROM을 사용하고 있는지 선택을 해 줍니다. 여기에서는 현재 장비에 장착되어 있는 ‘Xilinx PROM’ 을 선택해 줍니다. 다음으로 PROM File Format 은 일반적인 PROM 프로그램 파일 형태인 “MCS”를 선택해 줍니다. Checksum Fill Value (2 Hex Digits)는 Default로 설정되어 있는 “FF”로 두고 다음에 있는 파일의 이름과 저장되는 폴더는 사용자가 직접 설정을 해 줍니다. 설정 전의 PROM File Name는 Untitled, Location은 프로젝트가 선언된 폴더로 지정이 되어 있습니다. 다음의 그림에서는 이러한 작업을 수행한 그림이 나와 있습니다.



- **Specify Xilinx PROM Device**

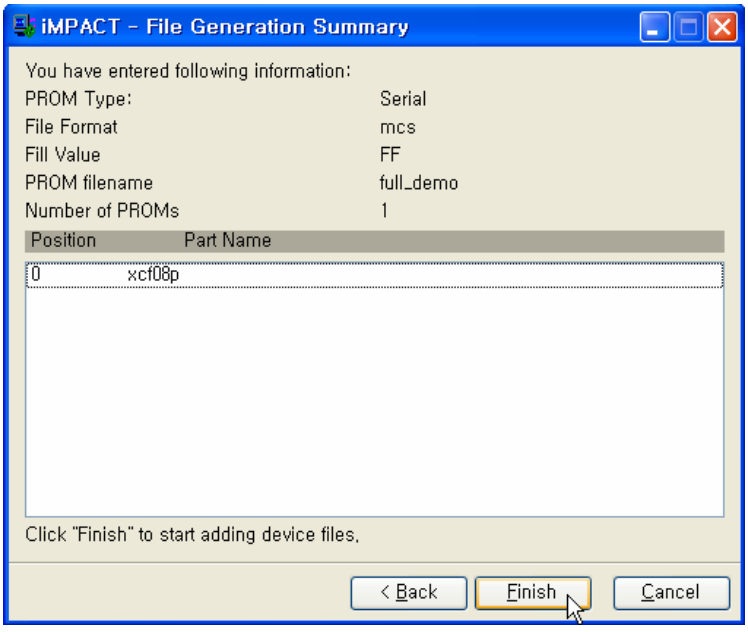
❑ 이 창에서는 현재 장비에서 사용하고 있는 디바이스인 Xilinx PROM을 선택하는 작업을 합니다. 따라서 현재 장비에서 사용하고 있는 PROM을 선택해 주고 이러한 PROM이 몇 개 사용하는지를 이 창에서 결정해 주는 것 입니다. 따라서 현재 창에서는 Select a PROM 에서 Xilinx Device Module 에서 사용하고 있는 PROM인 XCF08PVO48 또는 XCF16PVO48을 선택하게 됩니다.

❑ 따라서 이러한 디바이스를 선택하고 현재 모듈에서 몇 개를 사용하는지를 “Add” 버튼을 이용하여 넣어 주면 됩니다. 여기에서는 현재 디바이스 모듈을 확인해 사용하는 디바이스를 넣어주고 가운데 창을 통해 디바이스 첨가해서 확인을 해 주면 됩니다.

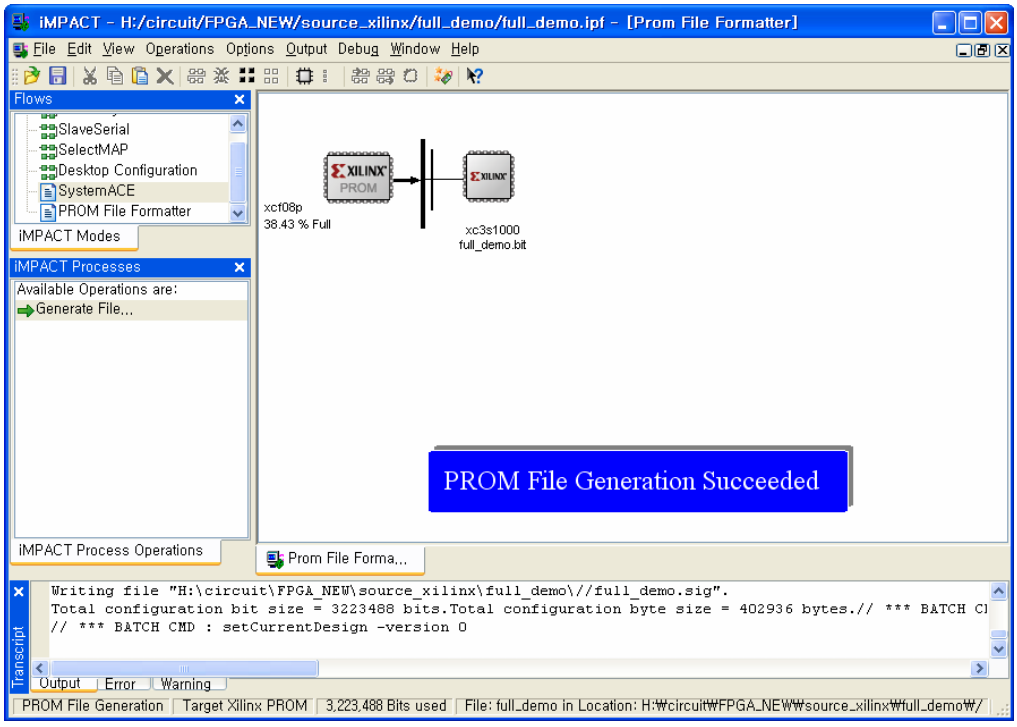


- **File Generation Summary**

❑ 여기에서는 이 제까지의 작업에 대한 Summary 형태로 보여주고 있습니다. 따라서 이전에서 작업한 것에 대한 내용을 이 창을 통해 최종적으로 확인해 볼 수 있습니다. 여기에서 확인한 내용이 이상이 없으면 Finish 버튼을 눌러 마치고, 수정 사항이 이상이 있을 시 Back 버튼을 통해 이 전 단계로 가서 다시 수정을 해 주면 됩니다.



이상의 작업을 마치고 아래의 Xilinx PROM에 적용을 해줄 파일 변환에 대한 창이 활성화 됩니다. 따라서 여기에서 변환할 .bit 파일을 현재의 작업 폴더에서 찾아서 넣어주면 됩니다. 이렇게 하고 창에서 마우스 오른쪽 버튼을 이용해 “Generate File...”을 이용해 파일 변환을 수행하면 됩니다. 아래에서 이러한 작업의 완료된 모습을 보여주고 있습니다.



현재까지 두 메인 디바이스에 대한 PROM 파일 변환 작업을 수행하였습니다. 사전에 위와 같이 파일 변환 작업을 하고 전용 다운로드 케이블을 이용하여 프로그램 수행을 하면 됩니다.

3) Configuration ROM을 이용한 FPGA Verify (Verify)

Configuration ROM에 저장된 프로그램 데이터를 이용하여 FPGA를 동작시키는 방법입니다. 이는 Configuration ROM에 미리 저장된 소스가 있을 시 이러한 Verify 동작이 실행이 됩니다. 이 동작은 초기에 전원을 켤 때, 미리 RROM에 저장된 데이터가 FPGA 디바이스로 전송되어 데모 소스가 실행이 되게 됩니다. 또한 FPGA device module에 있는 ‘Reset’ 버튼을 이용한 Verify 동작을 수행해 줄 수 있습니다. 만약 PROM에 어떠한 데이터가 없을 때, Power ON 동작 및 ‘Reset’ 버튼에 의한 동작은 실행되지 않습니다.

03
HBE-COMBO II
User's Manual &
Lab Guide

제품 사용하기

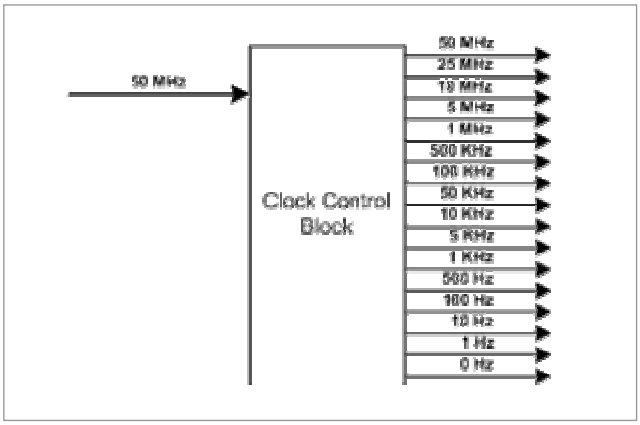
3. 제품 사용하기

3.1 Clock Control Block

1) 구성

장비 내부에는 Clock control block이 있어서 사용자가 원하는 클럭을 간단한 스위치의 조작으로 입력 받아 사용할 수 있습니다. 이 제어 블록은 오실레이터로 입력되는 50MHz의 클럭을 별도의 CPLD에서 16분주하여 FPGA 디바이스 모듈의 입력 전용 핀으로 분주된 클럭이 입력되고 있습니다. 따라서 사용자는 Clock Control Switch를 이용하여 16개의 클럭을 선택할 수 있습니다. Clock Control Block의 구성은 50 MHz의 오실레이터와 표시 부, 클럭 제어부의 3부분으로 구성하고 있습니다.

오실레이터 부는 전원 공급으로 주파수를 생성하는 장치로 장비에서 필요한 클럭을 생성하고 있습니다. 표시 부는 7-Segment와 LED로 구성하고 있고, 7-Segment에서는 주파수 값이 LED에는 주파수 대역(MHz, KHz, Hz)을 표시하게 됩니다. 클럭 제어부는 오실레이터에서 나오는 값을 클럭 제어 스위치에 따라 16분주하여 FPGA 디바이스 모듈로 전달하는 역할을 하고 있습니다.



[그림 3-1] Clock Control Block 구성

2) 동작

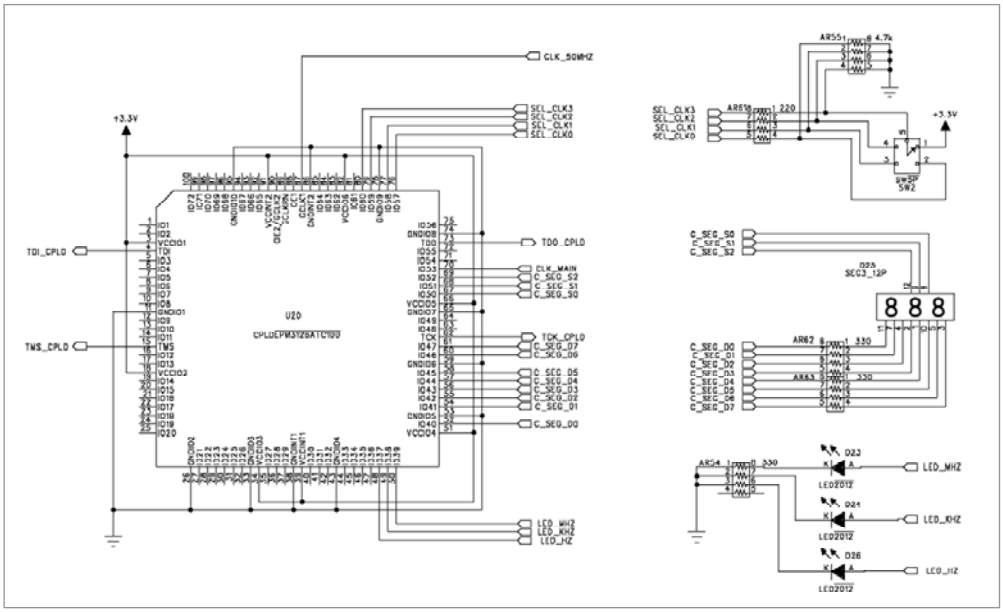
장비에서 사용하는 클럭은 기본으로 Base board 에 있는 50 MHz를 가지고 사용하게 됩니다. 따라서 클럭 제어 스위치를 조절하면 7-Segment에 현재 FPGA 디바이스 모듈로 공급되는 클럭 값이 표시되고 오른 쪽에 있는 LED를 통해 현재 주파수 대역을 확인할 수 있습니다.

이 스위치를 조절하여 0 Hz ~ 50 MHz 의 분주된 클럭의 값을 FPGA 디바이스 모듈로 전달되게 됩니다. 또한 FPGA 디바이스 모듈에는 별도로 오실레이터를 꼽아 사용할 수 있는 소켓을 구성해 놓았습니다. 이 소켓에는 사용자가 원하는 값을 가진 오실레이터를 장착해서 사용할 수 있습니다. 따라서 이 자리에 오실레이터를 장착하고 User Clk EN 스위치를 On 하여 사용하면 됩니다. Main clock 과 User clock은 서로 다른 입력 전용 핀으로 연결이 되어 있으므로 설계 소프트웨어에서 핀 연결을 하여 사용하면 됩니다.

오실레이터의 출력 클럭은 다음에서 확인할 수 있습니다. 표에서는 Clock Select 스위치에 따른 출력 주파수를 확인할 수 있습니다. 장비에서는 출력 클럭의 확인은 3개의 Segment를 이용하여 주파수 값을 확인할 수 있고, Segment 옆에 있는 3개의 LED를 통해 주파수 대역을 확인 할 수 있습니다. 이처럼 장비에서는 손쉽게 원하는 클럭을 제어하여 FPGA 디바이스로 입력할 수 있습니다.

Clock SW	클럭 입력	Clock SW	클럭 입력	Clock SW	클럭 입력	Clock SW	클럭 입력
0	0Hz	4	100Hz	8	10kHz	C	1MHz
1	1Hz	5	500Hz	9	50kHz	D	5MHz
2	10Hz	6	1kHz	A	100kHz	E	25MHz
3	50Hz	7	5kHz	B	500kHz	F	50MHz

3) 회로도



4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
CLK_MAIN[0]	N1	A13	Main Clock Input
CLK_MAIN[1]	A13	C14	Main Clock Input
CLK_USER[0]	N2	B13	User Clock Input
CLK_USER[1]	B13	B14	User Clock Input

3.2 VFD (16 x 2 Line)

1) VFD란

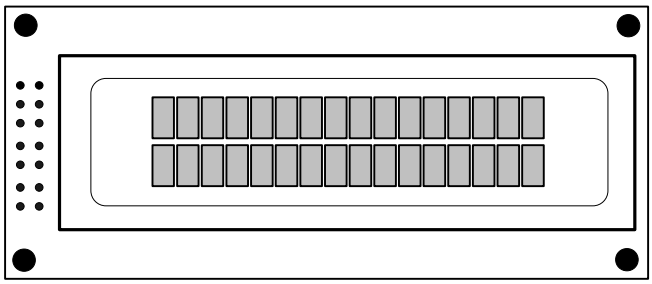
VFD[Vacuum Fluorescent Display]는 형광 표시 판으로써 1960년도에 개발되어 전자계산기에 적용되기 시작 하였으며, 80년대에 다색화에 성공하여 여러 색깔을 만들어 가고 있습니다. 이 모듈은 CRT의 전자총 역할을 하는 필라멘트에서 항시 방출되는 전자를 형광체에 충돌시켜 자체적으로 발광시키는 소자입니다. 따라서 예전의 Text LCD에서 보는 백 라이트에서 보는 것과 달리 하나의 도트마다 자체 발광하여 출력하는 장치라 할 수 있습니다.

2) 특징

VFD는 형광 물질의 조절로 여러가지 색의 표시가 용이하고, 자체적으로 발광하기 때문에 시인성이 뛰어납니다. 또한 LED등과 비교 시 빛이 부드러워 눈의 피로를 덜고 어두운 곳이나 밝은 곳에서도 문자 식별이 가능하며, 시야 각 또한 우수하고 저 전압으로 구동이 가능하여 부품의 적용이 쉬우며 고 신뢰성을 가지고 있습니다.

3) 구성

VFD 모듈의 외부 구조는 아래의 그림과 같이 16문자 X 2의 표시부를 가지며 표시부 좌측에 커넥터가 위치한 형태입니다.



[그림 3-2] 16 x 2 VFD

사용되는 인터페이스 커넥터 핀의 기능을 요약하면 다음의 표와 같습니다.

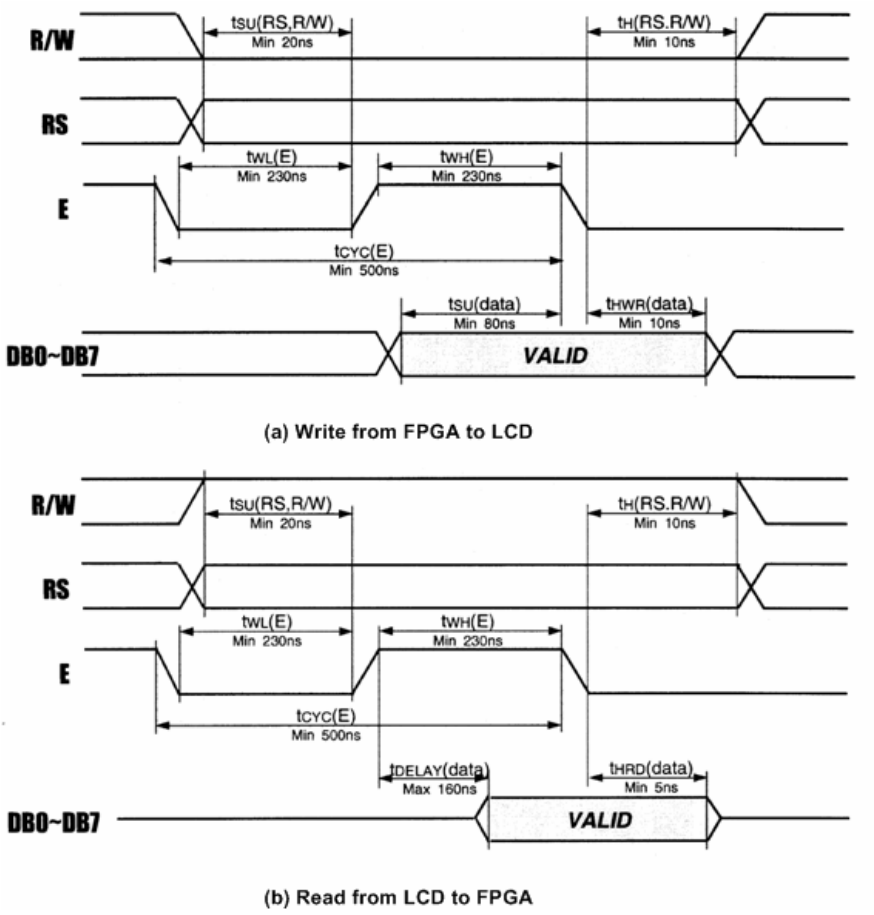
〈표 3-1〉 VFD 인터페이스 핀

핀	신호명	기 능
1	GND	전원 GND
2	V_{CC}	전원 +5V
3	/RST	Reset input
4	RS	Register Select (0 = instruction, 1 = data)
5	R/W	Read/Write (0 = FPGA -> LCD, 1 : FPGA <- LCD)
6	E	Enable Signal for read/write LCD
7	DB0 (LSB)	
8	DB1	
9	DB2	
10	DB3	
11	DB4	14 13

12	DB5	
13	DB6	
14	DB7	

4) 동작 타이밍

VFD 모듈은 FPGA로 접속할 때는 각 제어신호의 동작 타이밍을 고려하여야 합니다. VFD 모듈은 과거의 Text LCD 보다 동작 타이밍에 대한 성능이 우수해 졌습니다. VFD 모듈의 read 및 write 동작시의 타이밍 도를 보면 아래의 그림과 같습니다.



[그림 3-3] VFD 모듈의 동작 타이밍

VFD 모듈을 사용하기 위한 제어 명령을 정리하면 다음 페이지와 표와 같습니다. 이들 명령은 FPGA가 VFD 모듈을 제어하는 프로그램에서 사용되며 데이터 버스 DB0~DB7

을 통하여 전송됩니다.

VFD 모듈이 각 명령을 받아 이를 실행하기 위해서는 지정된 시간이 필요하므로 FPGA는 그 다음의 명령을 보내기 전에 충분히 대기하거나 busy flag을 조사하여 앞에서 전송한 제어 코드의 실행이 완료되었는지의 여부를 확인하여야 합니다.

〈표 3-2〉 VFD모듈 제어명령

Instruction	CODE									
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Display Clear	0	0	0	0	0	0	0	0	0	1
Cursor Home	0	0	0	0	0	0	0	0	1	x
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B
Cursor/Display Shift	0	0	0	0	0	1	S/C	R/L	x	x
Function Set	0	0	0	0	1	IF	N	x	BR1	BR0
CGRAM Address Setting	0	0	0	1						
DDRAM Address Setting	0	0	1							
Busy Flag & Address Reading	0	1	BF							
Busy Writing to CG or DDRAM	1	0								
Data Reading from CG or DDRAM	1	1								

NOTE	I/D = 1 : Increment I/D = 0 : Decrement S = 1 : Display shift enabled S = 0 : Cursor shift enabled S/C = 1 : Display shift S/C = 1 : Cursor move R/L = 1 : Shift to the right R/L = 0 : Shift to the left IF = 1 : 8bit IF = 0 : 4bit N = 1 : 2Lines display N = 0 : 1 Line display BR1, BR0 = 00 : 100% 01 : 75% 10 : 50% 11 : 25% BF = 1 : Busy (Internally operation) BF = 0 : Not busy (Instruction acceptable)	[Abbreviation] DD-RAM : Display Data RAM CG-RAM : Character Generator RAM ACG : CG-RAM Address ADD : DD-RAM Address ACC : Address Counter

❑ Display Clear

▶ 전체 화면을 지우고 어드레스 카운터를 DD-RAM 어드레스 0으로 하여 커서를 home 위치로 합니다.

❑ Cursor home

▶ DD RAM의 내용은 변경하지 않고 커서만을 home 위치로 합니다.

❑ Entry mode set

▶ 데이터를 read하거나 write할 경우에 커서의 위치를 증가시킬 것인가(I/D=1) 감소 시킬 것인가 (I/D=0)를 결정하며, 또 이때 화면을 시프트 할 것인지(S=1) 아닌지(S=0)를 결정합니다.

❑ Display ON/OFF control

▶ 화면 표시를 ON/OFF 하거나(D) 커서를 ON/OFF하거나(C) 커서를 깜박이게 할 것인지(B)의 여부를 지정 합니다.

❑ Cursor/Display shift

▶ 화면(S/C=1) 또는 커서(S/C=0)를 오른쪽(R/L=1) 또는 왼쪽(R/L=0)으로 시프트 합니다.

❑ Function set

▶ 인터페이스에서 데이터의 길이를 8비트(DL=1) 또는 4비트(DL=0)로 지정하고, 화면 표시 행수를 2행(N=1) 또는 1행(N=0)으로 지정하며, 화면의 밝기를 BR1~0에 의해 100%, 75%, 50%, 25%로 설정을 해 줄 수 있습니다.

※ 전원 투입 후 최초에는 주로 이 명령을 보내게 됩니다. 또한, 4비트로 인터페이스 할 경우에는 DB4~DB7을 사용하며, 상위 4 bit를 먼저 전송하고 다음에 하위 4 bit를 전송해야 합니다.

❑ CGRAM address Setting

▶ Character Generator RAM의 어드레스를 지정합니다. 이후에 송수신하는 데이터는 CG RAM의 데이터입니다.

❑ DDRAM address Setting

▶ Display Data RAM의 어드레스를 지정합니다. 이후에 송수신하는 데이터는 DD RAM의 데이터입니다.

❑ Busy flag & address Reading

▶ LCD 모듈이 내부 동작중임을 나타내는 Busy Flag(BF) 및 어드레스 카운터의 내용을 read 합니다. LCD 모듈이 각 제어 코드를 실행하는데 지정된 시간이 필요하므로 FPGA가 BF를 읽어 1일 경우에는 기다리고 0일 경우 다음 제어 코드를 보내는 방법을 사용하면 보다 효율적인 처리가 가능 합니다.

❑ Data Writing to CG or DDRAM

▶ CG-RAM 또는 DDRAM에 데이터를 쓰는 동작을 합니다.

❑ Data Reading from CG or DDRAM

▶ CG-RAM 또는 DDRAM에 데이터를 읽는 동작을 합니다.

VFD 모듈을 전원 투입부터 초기화하는 과정을 요약하면 다음과 같습니다.

- 전원을 투입 합니다.
- VFD 모듈이 리셋 되려면 약 50ms가 소요되므로 이 시간 이상을 기다립니다.
- Function set 명령(001xxx00)을 보냅니다.
- Display ON/OFF control 명령(00001xxx)을 보냅니다.
- Entry mode set 명령(000001xx)을 보냅니다.
- DDRAM address를 보냅니다.
- 문자 데이터를 연속으로 보냅니다.
- 필요에 따라 위의 과정을 반복 합니다.

5) DD RAM 어드레스

DD RAM은 표시될 각 문자의 ASCII 코드 데이터가 저장되어 있는 메모리이며 모두 80개의 번지가 있는데, 화면의 각 행과 열의 위치에는 고유한 Address 값이 부여됩니다. 그런데, 각 행과 행 사이의 어드레스가 연속하여 있지 않으므로 주의하여야 한다. 표시 문자의 위치에 대한 DD RAM의 어드레스는 <표 3-3>과 같습니다.

〈표 3-3〉 표시문자 위치에 대한 DD RAM 어드레스

	1															16
제1행	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
제2행	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

6) 표시 문자 세트

VFD 모듈에서 화면에 표시할 수 있는 문자의 종류에는 대부분의 ASCII 도형문자들이 포함되며, 기타 일본어의 카타가나 문자와 몇 가지 특수문자들이 포함됩니다.

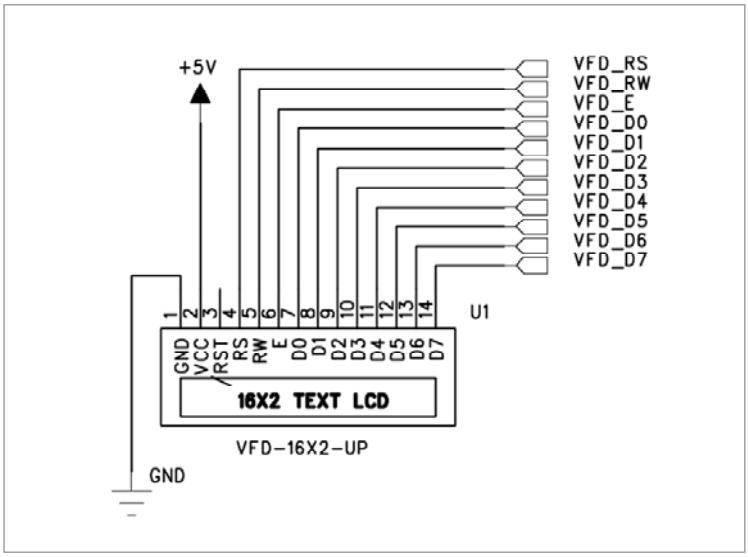
00H~0FH의 영역에는 사용자 정의문자를 설정하여 사용할 수 있다. 여기에는 최대 8문자를 정의할 수 있는데, 이에 대한 정보는 전원이 인가된 후에 정의되어야 하며, 전원이 꺼지면 정의된 내용은 기억되지 않습니다.

VFD 모듈에서 화면에 표시할 수 있는 문자 중 ASCII 도형문자의 종류와 코드 값을 보면 아래의 표와 같습니다.

〈표 3-4〉 ASCII 도형문자의 종류 및 코드

Upper bits / Lower bits					DB7	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					DB6	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					DB5	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					DB4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
DB3	DB2	DB1	DB0			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	CG-RAM (1)																
0	0	0	1	1	CG-RAM (2)																
0	0	1	0	2	CG-RAM (3)																
0	0	1	1	3	CG-RAM] (4)																
0	1	0	0	4	CG-RAM (5)																
0	1	0	1	5	CG-RAM (6)																
0	1	1	0	6	CG-RAM (7)																
0	1	1	1	7	CG-RAM (8)																
1	0	0	0	8	CG-RAM (1)																
1	0	0	1	9	CG-RAM (2)																
1	0	1	0	A	CG-RAM (3)																
1	0	1	1	B	CG-RAM] (4)																
1	1	0	0	C	CG-RAM (5)																
1	1	0	1	D	CG-RAM (6)																
1	1	1	0	E	CG-RAM (7)																
1	1	1	1	F	CG-RAM (8)																

7) 회로도



VFD 모듈은 총 14개의 핀으로 구성되어 있습니다. 여기에는 전원 및 데어터 전송, 컨트롤 핀, reset핀으로 구성되어 있습니다. 회로도에 나와 있는것과 같이 전원은 보드에서 공급받아 사용할 수 있도록 구성 되어 있고, Reset 핀은 하드웨어적으로는 사용하고 있지 않습니다. VFD의 사요을 위해서는 제어신호인 VFD_RS, VFD_RW, VFD_E의 제어신호와 데이터 신호 8개를 FPGA에서 직접 제어하여 VFD를 Display 하게 됩니다.

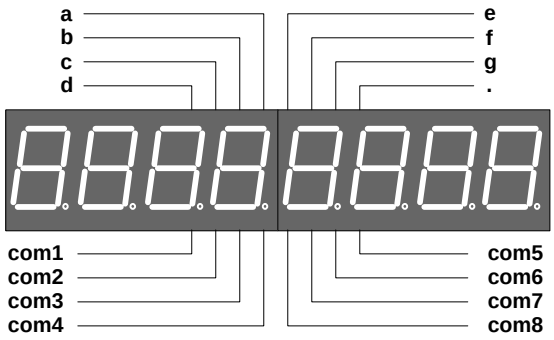
8) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
VFD_D[0]	R7	R6	VFD DATA 0
VFD_D[1]	R6	R5	VFD DATA 1
VFD_D[2]	P3	P4	VFD DATA 2
VFD_D[3]	R8	P3	VFD DATA 3
VFD_D[4]	P6	P2	VFD DATA 4
VFD_D[5]	P4	P1	VFD DATA 5
VFD_D[6]	P9	P8	VFD DATA 6
VFD_D[7]	P7	P7	VFD DATA 7
VFD_E	R4	R7	VFD Enable
VFD_RW	R5	R8	VFD Read/Write
VFD_RS	R2	T8	VFD Register Select

3.3 7-Segment Array

1) 구성

본 제품에서 사용된 4개의 7-Segment가 하나로 구성되어 있는 7-Segment LED Array 두 개를 묶어 다음 그림과 같은 구성으로 이루어져 있습니다.

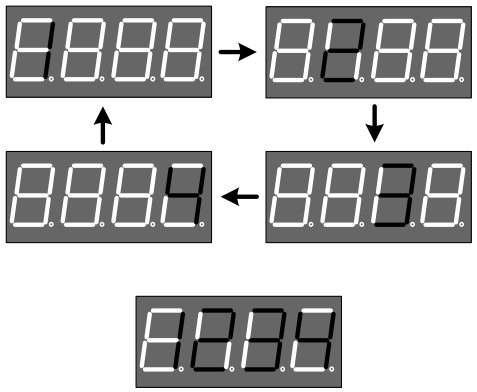


[그림 3-4] 7-Segnent LED Array 구성

2) 동작

일반적인 방법에서의 7-Segment라면 8개를 사용할 경우 하나의 7-Segment당 출력인 'a','b','c','d','e','f','g','.'가 각각 출력 핀을 가져 64개의 출력 핀이 필요로 하게 됩니다. 이러 할 경우 FPGA의 한정된 핀을 가지고 Segment에 너무 많은 핀이 할당되게 되므로 핀 소모를 개선하기 위하여 위와 같이 7-Segment의 Data 라인은 공통으로 연결되어 있고 출력할 위치인 7-Segment를 지정하는 com1 ~ com8의 값을 제어하여 원하는 숫자나 문자를 표시하는 방법을 사용하게 됩니다.

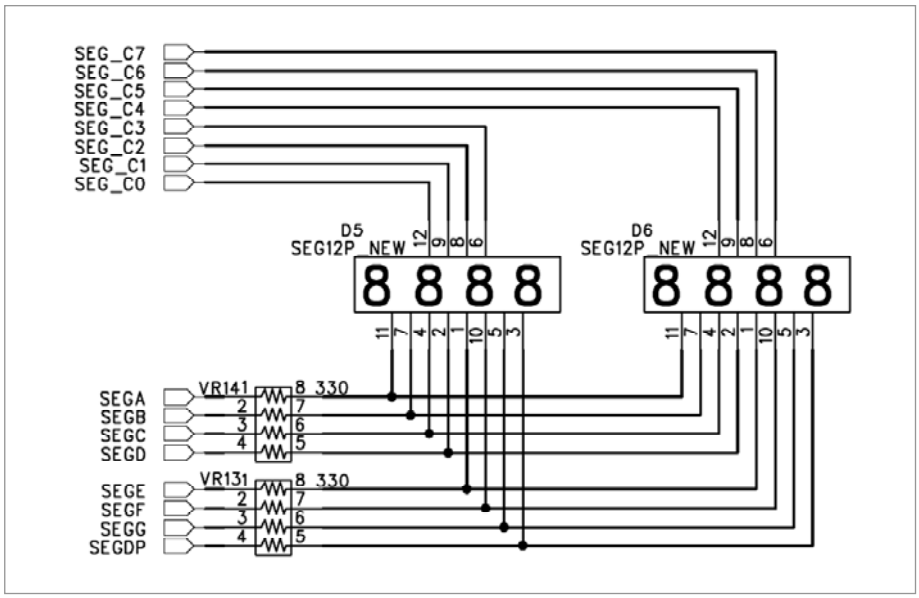
이렇게 하면 16개의 핀을 가지고 8개의 7-Segment를 제어하게 됩니다. 따라서 8개의 7-Segment 중에 Display 하려는 Segment common 핀을 선택하고 동시에 Segment에 Data를 주는 형태로 7-Segment를 제어할 수 있다. 아래의 그림은 4개의 7-Segment에 "1234"의 숫자를 표시하기 위한 방법을 설명한 것입니다.



다시 설명하면 7-segment 데이터 값을 “00000110”로 주어 ‘1’를 표시하는 값을 주고 com1에 ‘0’을 그리고 나머지 com2~4는 ‘1’의 값을 주면 첫째 7-Segment에 ‘1’이 표시됩니다. 다음으로 데이터에 “01011011”을 주어 ‘2’의 값을 주고 com2에 ‘0’과 com1, com3~4에 ‘1’을 주면 둘째 7-Segment에 2가 표시됩니다.

이런 순서로 4까지의 숫자를 표시하고 다시 처음으로 돌아가 위의 내용을 반복합니다. 이를 약 1ms 이상의 주기로 반복하면 잔상효과에 의해 “1234”의 숫자가 모두 켜져 있는 것처럼 표시되게 됩니다. 이러한 형태로 8개의 7-segment를 com 핀을 이용하여 선택하고, 해당되는 data 값을 줌으로써 각각의 7-Segment의 제어가 가능하게 됩니다.

3) 회로도



이전에 설명 했듯이 회로도 에서는 16개의 핀을 가지고 7개의 Segment를 제어하고 있습니다. 위쪽에 있는 SEG_C의 8개의 핀을 이용하여 각각의 Segment를 선택하게 됩니다. 그리고 데이터 핀인 SEGA~SEGDP의 핀을 제어하여 7-Segment를 제어하게 됩니다. 이 모든 작업은 FPGA 디바이스와 연결된 I/O핀에서 제어를 하면 됩니다.

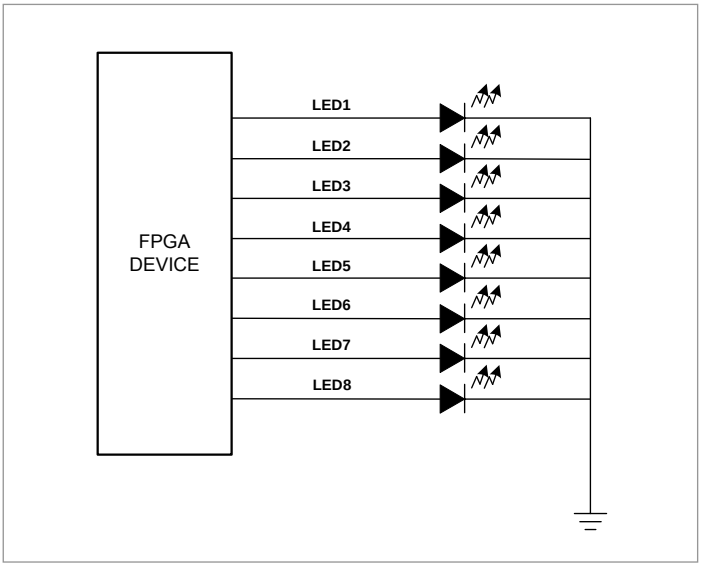
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
SEG_COM[0]	Y1	AD2	Segment 1 Select
SEG_COM[1]	Y4	W2	Segment 2 Select
SEG_COM[2]	Y3	A1	Segment 3 Select
SEG_COM[3]	W1	AB4	Segment 4 Select
SEG_COM[4]	Y5	AB3	Segment 5 Select
SEG_COM[5]	W3	W6	Segment 6 Select
SEG_COM[6]	W2	W5	Segment 7 Select
SEG_COM[7]	V1	W4	Segment 8 Select
SEG_DATA[0]	AF5	AF5	Segment data A
SEG_DATA[1]	AE5	AE5	Segment data B
SEG_DATA[2]	AD6	AB6	Segment data C
SEG_DATA[3]	AC6	AA6	Segment data D
SEG_DATA[4]	AA2	K26	Segment data E
SEG_DATA[5]	AA1	K25	Segment data F
SEG_DATA[6]	AA6	AC2	Segment data G
SEG_DATA[7]	AA5	AC1	Segment data H

3.4 LED

1) 구성

LED는 Bit 출력을 확인하거나 동작의 상태를 표시하기 위해 사용하는데 본 제품에서는 8개의 DIP 타입의 LED를 제공하고 있습니다. 또한 사용자가 보는 시인성을 좋게 하기 위해 고휘도 LED를 사용하고 있습니다. 아래 그림은 LED에 대한 블록도를 보여주고 있습니다.

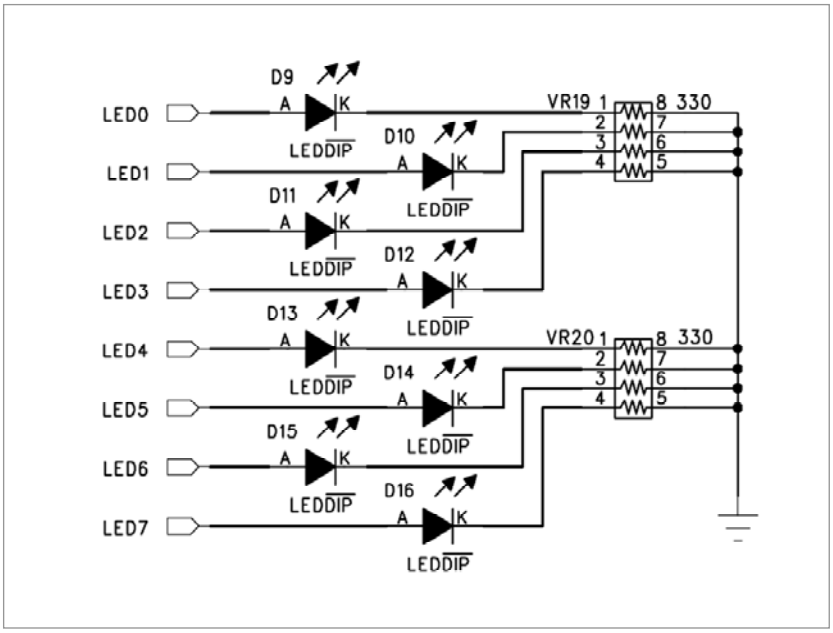


[그림 3-5] LED 블록도

2) 동작

LED의 입력 단에 디지털 신호 '1'에 해당하는 입력이 인가 될 경우 LED에 불이 들어오게 됩니다. 초기의 전원 인가 시 LED의 특성상 미세한 전류의 흐름에 의해서도 희미하게 LED가 켜지는 경우가 있는데 이것은 제품의 문제가 아니며 프로그램 되지 않는 디바이스의 잔류 전류가 나와서 이러한 현상이 일어나게 됩니다. 이러한 증상은 툴을 통한 디바이스를 프로그램을 끝내고 디바이스가 동작을 시작하면 LED는 정상적으로 동작하게 됩니다.

3) 회로



LED는 회로도 에서 보는 것과 같이 8개의 LED가 각각의 FPGA의 I/O 핀과 시리얼 저항에 연결 되어 있음을 볼 수 있습니다. 따라서 FPGA에 연결된 부분에 '1'이라는 신호를 주어 LED를 제어 할 수 있도록 구성 하였습니다.

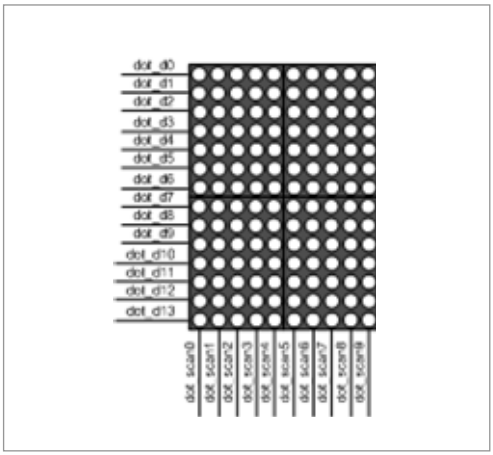
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
LED_D(0)	AF7	AB7	D1 LED display
LED_D(1)	AE7	AA7	D2 LED display
LED_D(2)	AB8	AE7	D3 LED display
LED_D(3)	W8	AC7	D4 LED display
LED_D(4)	AF6	AD6	D5 LED display
LED_D(5)	AE6	AC6	D6 LED display
LED_D(6)	AD7	AF6	D7 LED display
LED_D(7)	AC7	AE6	D8 LED display

3.5 Dot matrix LED

1) 구성

본 제품에 사용된 Dot matrix LED 모듈은 5열x7행dot LED 4개를 조합하여 10개의 열과 14개의 행으로 구성하고 있습니다. 이는 영문, 숫자 및 한글까지 표현할 수 있도록 구성할 것입니다. Dot matrix LED 모듈의 구성은 아래의 그림과 같습니다.



[그림 3-6] Dot matrix LED 구성

Dot_d0~dot_d13은 해당 열에 표시될 데이터 입력이 되며 데이터 값은 '1'의 값을 가질 경우 켜지게 됩니다. dot_scan0~dot_scan9는 dot_d0~dot_d13의 데이터를 표시할 열의 위치를 지정하는 입력이며 '1'의 값을 가질 때 해당 열이 선택됩니다.

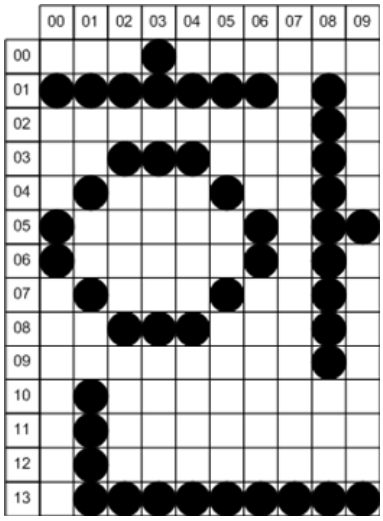
예를 들어 1행 1열의 dot LED를 켜기 위해서는 dot_d0의 값을 '1'로 입력하고 dot_scan0의 값을 '1'으로 할 때 1행 1열의 dot LED가 켜지게 됩니다. 이런 방법으로 dot의 scan 라인을 선택하고, 해당되는 라인에 data값을 주어서 dot를 display하게 됩니다. 이것도 7-Segment와 같이 1ms 이상의 주기로 반복해서 display 하게 되면 눈의 잔상 효과로 인해서 모든 dot가 display 되어 있는 것처럼 느끼게 됩니다. 이러한 구동 방식은 7-Segment의 scan 방식과 비슷한 구조를 가지고 있습니다.

2) 동작

다음은 아래 그림의 한글과 영문을 동시에 display하기 위한 방법에 대해 살펴보겠습니다.

처음으로 dot_d13~dot_d0에 "00000001100010"인 첫째 열의 데이터를 입력하고 dot_scan9~ dot_scan0에 "1111111110"의 값을 입력하면 첫째 열의 원하는 dot LED가 켜지

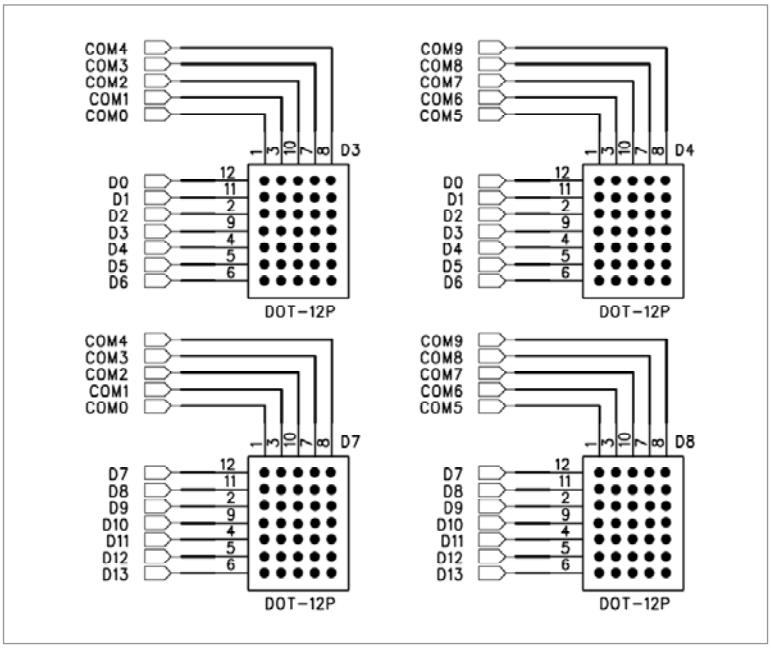
게 됩니다. 둘째로 dot_d13~dot_d0에 “11110010010010”인 둘째 열의 데이터를 입력하고 dot_scan9~dot_scan0에 “1111111101”의 값을 입력하면 둘째 열의 원하는 dot LED가 켜지게 됩니다. 이를 열 다섯 번째 열까지 반복하며, 반복되는 주기를 약 1m초 이하로 해서 반복하게 되면 눈의 잔상효과에 의해서 ‘한’이라는 글자가 dot matrix LED 모듈에 표시됩니다.



이를 정리하면 다음과 같습니다.

구 분	dot_d13~dot_d0	dot_scan9~dot_scan0
0 열	“00000001100010”	“1111111110”
1 열	“11110010010010”	“1111111101”
2 열	“10000100001010”	“1111111011”
3 열	“10000100001011”	“1111110111”
4 열	“10000100001010”	“1111101111”
5 열	“10000010010010”	“1111011111”
6 열	“10000001100010”	“1110111111”
7 열	“10000000000000”	“1101111111”
8 열	“10001111111110”	“1011111111”
9 열	“10000000100000”	“0111111111”

3) 회로



회로에서 보는 것과 같이 5 X 7의 Dot LED 4개를 사용하고 있고, 행과 열의 데이터 라인을 공통으로 사용하고 있는 모습을 볼 수 있습니다. 따라서 행과 열의 신호가 서로 일치되는 구간에서 Dot의 LED에 불이 들어 오게 됩니다. 현재 장비에서는 동작을 스캔 라인과 데이터 라인으로 구분 지어 동작 시키고 있습니다. 따라서 스캔 라인을 하나씩 일정한 주기로 시프트 시키고, 해당되는 스캔 라인에 데이터를 입력하는 방식으로 되어 있습니다.

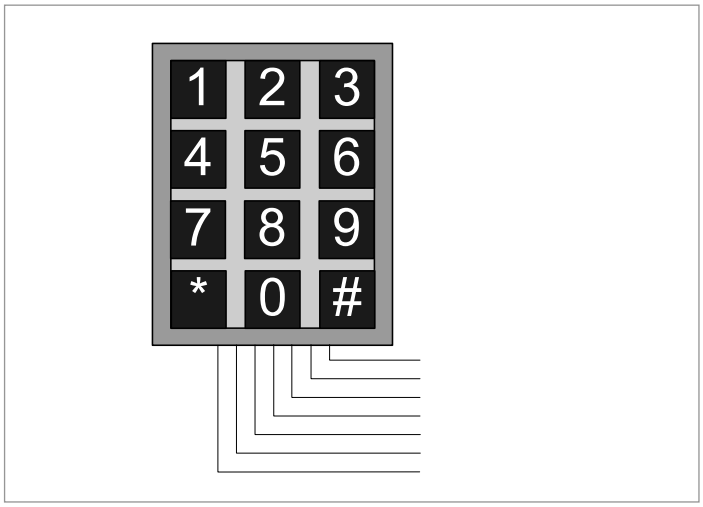
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
DOT_D[0]	U3	U4	DOT DATA 0
DOT_D[1]	U6	U3	DOT DATA
DOT_D[2]	U5	U2	DOT DATA
DOT_D[3]	U9	T2	DOT DATA
DOT_D[4]	U7	T1	DOT DATA
DOT_D[5]	T2	U7	DOT DATA
DOT_D[6]	U10	U6	DOT DATA
DOT_D[7]	T6	T7	DOT DATA
DOT_D[8]	T3	T6	DOT DATA
DOT_D[9]	T8	T5	DOT DATA
DOT_D[10]	T7	T4	DOT DATA
DOT_D[11]	T10	R3	DOT DATA
DOT_D[12]	T9	R2	DOT DATA
DOT_D[13]	R3	R1	DOT DATA
DOT_COM[0]	W4	W3	DOT COM LINE 1
DOT_COM[1]	V3	V4	DOT COM LINE 2
DOT_COM[2]	V2	V3	DOT COM LINE 3
DOT_COM[3]	V5	V2	DOT COM LINE 4
DOT_COM[4]	V4	W7	DOT COM LINE 5
DOT_COM[5]	V7	U1	DOT COM LINE 6
DOT_COM[6]	V6	V7	DOT COM LINE 7
DOT_COM[7]	U2	V6	DOT COM LINE 8
DOT_COM[8]	U1	V5	DOT COM LINE 9
DOT_COM[9]	U4	U5	DOT COM LINE 10

3.6 Keypad

1) 구성

입력으로 구성되는 첫 번째의 장치로 키 패드가 있습니다. 아래의 그림에서는 이러한 Keypad의 구성을 보여주고 있습니다. 소스로 사용되는 스위치는 4x3 키 패드와 6개 Button 스위치, 16개의 단자로 구성된 Dip 스위치로 구성되어 있습니다.



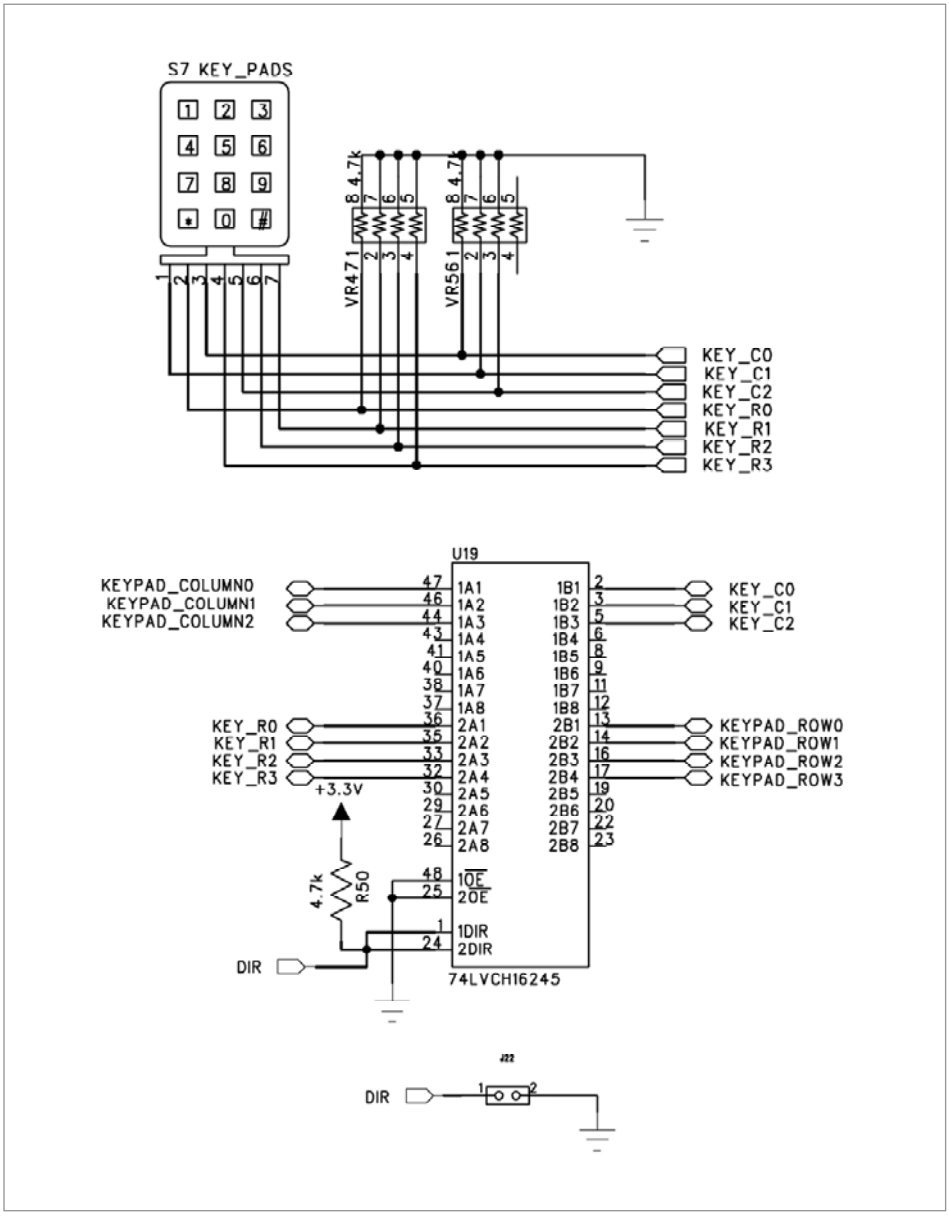
[그림 3-7] Keypad 구성

2) 동작

Keypad는 3x4의 12개의 키를 입력 받아서 사용할 수 있도록 구성해 놓았습니다. 이러한 키입력 방식은 스캔 방식 구조로써 가로 라인과 세로 라인에서 동시에 제어하여 입력을 받아 사용할 수 있도록 구성하고 있습니다. 현재 장비에서는 기본적으로 Keypad의 common 핀을 스캔으로 구동 시키게 구성되어 있습니다. 따라서 3개의 common핀을 FPGA 디바이스에서 스캔 형식으로 1의 값을 시프트 시키면서 해당되는 row 값에 입력되는 값을 보고 Keypad에 어떠한 값이 입력되었는지 알 수 있도록 구성되어 있습니다.

현재의 Keypad의 입출력 방향은 Scan Dir의 헤더 핀을 이용하여 바꿀 수 있습니다. 따라서 common을 FPGA 디바이스에서 스캔 형식으로 구동 할지, row 핀을 스캔 형식으로 구동 시킬 지는 Scan Dir에 점퍼 캡을 이용하여 제어할 수 있습니다.

3) 회로



회로에서 보는 것과 같이 3x4의 Keypad의 12개의 버튼을 입력 받아 사용할 수 있는 장치를 사용하고 있습니다. 여기에는 모두 풀다운 저항을 사용하고 있어서 모든 데이터 라인에 기본적으로 '0'의 값을 가지고 있습니다. 여기에서는 양방향 버퍼를 사용하여 common과 row의 데이터 라인의 입, 출력 방향을 바꿀 수 있습니다. 따라서 현재

기본으로 나가는 common의 입력형태를 Scan Dir의 점퍼 캡을 이용으로 출력으로 바꾸어서 사용이 가능합니다.

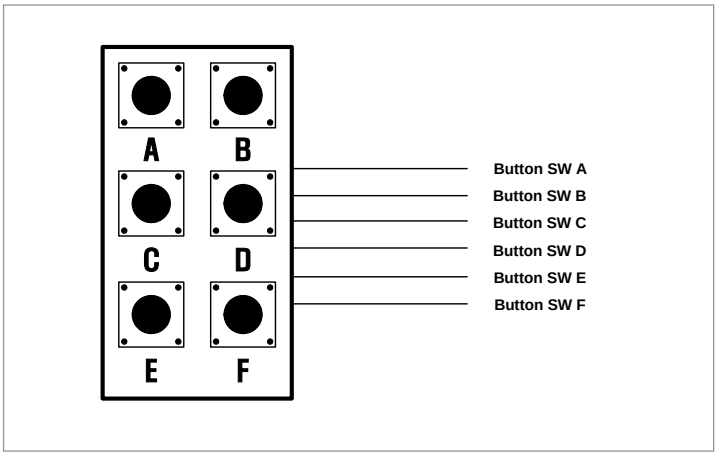
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
KEYPAD_COLUMN[0]	V11	Y12	Column Data 0
KEYPAD_COLUMN[1]	AB10	Y11	Column Data 1
KEYPAD_COLUMN[2]	AA10	W11	Column Data 2
KEYPAD_ROW[0]	AD10	AD10	Row Data 0
KEYPAD_ROW[1]	AC10	AC10	Row Data 1
KEYPAD_ROW[2]	V10	AB10	Row Data 2
KEYPAD_ROW[3]	AF9	AA10	Row Data 3

3.7 Button Switch

1) 구성

버튼 스위치는 6개로 구성이 되어 있으며 각 버튼 스위치가 별도의 FPGA I/O와 연결이 되어 있습니다. 아래의 그림은 이러한 Button Switch의 블록도를 보여주고 있습니다.



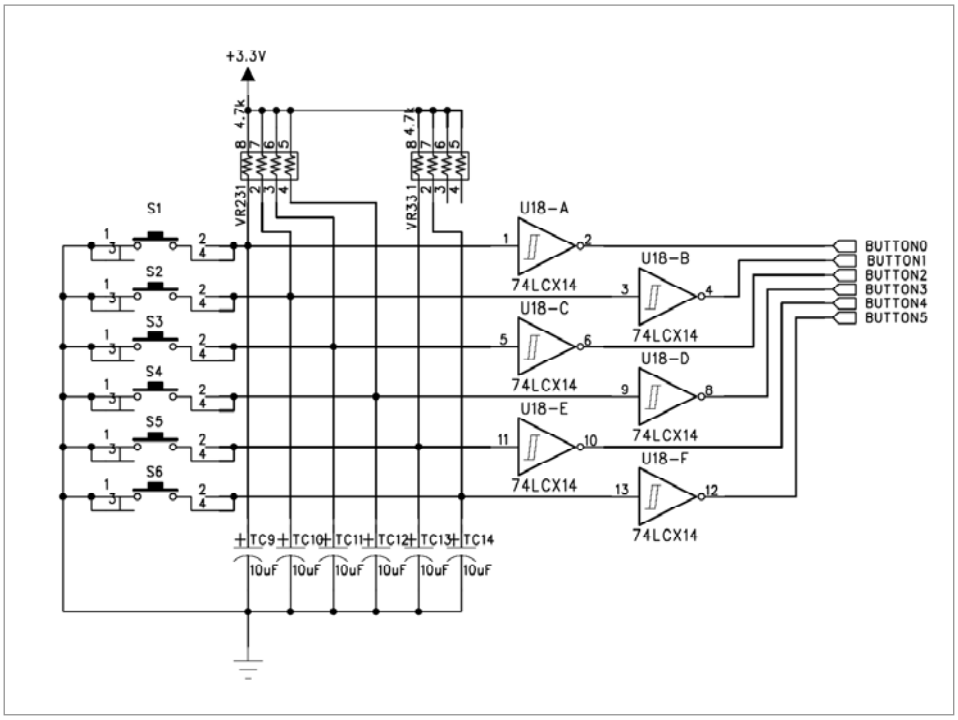
[그림 3-8] Push Button Switch 구성

2) 동작

버튼 스위치마다 별개의 I/O로 직접 연결이 되어 있으므로 Keypad 보다 제어를 간단하게 할 수 있습니다. 버튼 스위치는 키가 눌려 질 때 발생하는 채터링 현상을 방지하기 위한 회로가 내장되어 정확한 입력이 가능하도록 구성하였습니다.

3) 회로

회로도에서 보는 것과 같이 저항과 캐패시터를 사용한 채터링 방지 회로가 내장되어 있습니다. 현재 회로에서는 스위치가 안 눌려 졌을 때, 저항의 풀 업 저항에 의해 +3.3V 즉 '1'의 값이 가지게 됩니다. 따라서 FPGA 쪽으로 가는 신호는 74LCX14에 의해 반전되어 '0'의 값이 FPGA 디바이스에 입력되게 됩니다. 반대로 스위치가 눌려 졌을 때는 신호선이 GND와 연결되어 스위치에서는 '0'의 값을 가지게 되고 FPGA 디바이스로 가는 최종 데이터는 74LCX14에 의해 '1'의 값이 됩니다.



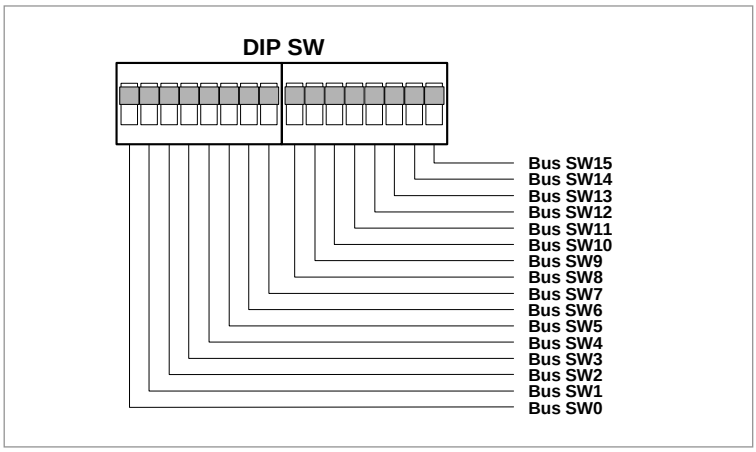
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
BUTTON_SW[0]	Y10	AD10	Detected High
BUTTON_SW[1]	W10	AC10	Detected High
BUTTON_SW[2]	AA9	AA9	Detected High
BUTTON_SW[3]	V9	Y9	Detected High
BUTTON_SW[4]	AE9	Y10	Detected High
BUTTON_SW[5]	AC9	AB9	Detected High

3.8 Bus Switch

1) 구성

버스 스위치는 입력의 경우 데이터 입력이나 모드 설정 등의 입력 소스로 주로 사용하게 됩니다. 여기에서는 8개의 딥 스위치를 2개 사용하여 총 16개의 버스 스위치를 제어 할 수 있는 구조를 가지고 있습니다. 아래 그림은 이에 대한 간단한 구성을 보여주고 있습니다.



[그림 3-9] Bus Switch 구성

2) 동작

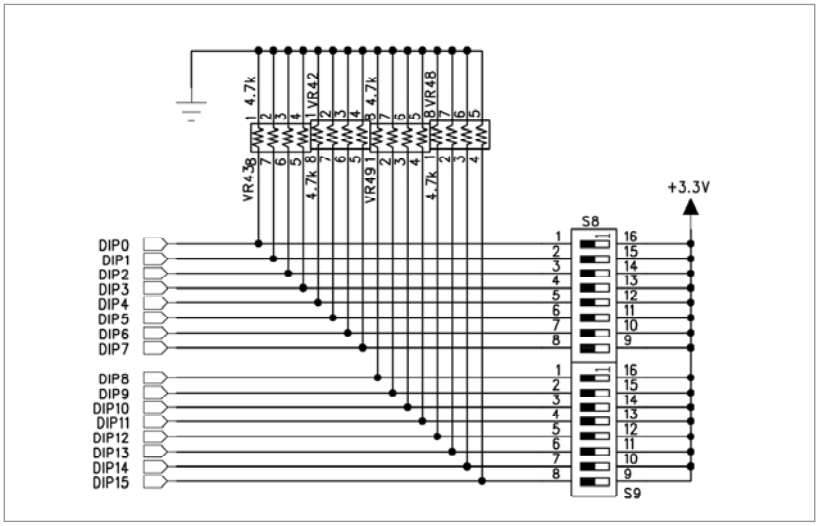
버스 스위치의 경우 데이터의 입력이나 모드 설정의 스위치로 동작을 하게 되므로 버튼의 설정이 고정 되는 형태를 취하고 있습니다. 동작은 스위치를 위로 올리면 디바이스 쪽에서 '1'의 값을 인식하고 내리면 반대로 '0'의 값이 전달 되는 동작을 하고 있습니다.

이 스위치는 버튼 스위치나 키 패드처럼 키가 리셋 되지 않고 그 상태를 유지를 하고 있기 때문에 사용하지 않을 때는 스위치를 OFF하여 디바이스에 필요치 않는 입력을 주지 않도록 하여 디바이스의 손상을 줄여야 합니다.

3) 회로

버스 스위치의 회로는 [그림 3-9]와 같이 구성이 되어 있습니다. 스위치의 OFF시 4.7K의 풀 다운 저항에 의해 '0'의 값이 FPGA 디바이스에 전달 되게 됩니다. 또한 스위치가 ON 되면 스위치에서 +3.3V와 연결이 되어 FPGA 디바이스로 '1'의 값이 전달 되어

HIGH값을 인식하게 됩니다.



▶ 스위치와 디바이스의 연결은 바로 연결 되는 것이 아니라 버퍼를 통해 디바이스의 I/O로 연결이 되어 있습니다. 이러한 것은 다른 출력 장치와 달리 스위치의 경우 FPGA 디바이스로 입력 신호를 보내는 역할, 즉 FPGA 디바이스는 스위치에서 신호를 입력 받아 사용하는 장치이므로 신호의 입력 시 전압이 불안정할 때 디바이스가 손상을 입을 수 있습니다. 따라서 버퍼를 통해 연결을 하여 이러한 문제점으로부터 디바이스의 손상을 줄이고 있습니다.

4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
DIP_D[0]	Y13	Y16	Bus Switch Data 0
DIP_D[1]	AB12	AB15	Bus Switch Data 1
DIP_D[2]	AA12	AA15	Bus Switch Data 2
DIP_D[3]	AD12	AE15	Bus Switch Data 3
DIP_D[4]	AC12	AD15	Bus Switch Data 4
DIP_D[5]	U12	AF13	Bus Switch Data 5
DIP_D[6]	AE11	AA13	Bus Switch Data 6
DIP_D[7]	Y12	W15	Bus Switch Data 7
DIP_D[8]	W12	AB14	Bus Switch Data 8
DIP_D[9]	AA11	AF12	Bus Switch Data 9
DIP_D[10]	Y11	AE12	Bus Switch Data 10
DIP_D[11]	AD11	Y13	Bus Switch Data 11
DIP_D[12]	AC11	W13	Bus Switch Data 12
DIP_D[13]	AF10	W12	Bus Switch Data 13
DIP_D[14]	AE10	AC11	Bus Switch Data 14

DIP_D[15]	W11	AD12	Bus Switch Data 15

3.9 Piezo

1) 구성

소리를 출력하는 장치로 흔히 사용하는 것이 스피커인데 스피커의 경우 소리의 높낮이와 주파수를 조정하여 원하는 소리를 출력할 수 있게 구성되어 있습니다. 이외에 사용할 수 있는 것이 Piezo인데 소리의 높낮이는 고정되어 있고 단지 소리의 주파수를 조정할 수 있도록 구성되어 있는 음성 출력 장치입니다. 장비에서는 FPGA I/O 핀 한 개와 연결 되어 있고, 이 핀에 주파수를 조절하여 주면 Piezo 소리를 조절할 수 있습니다.

2) 동작

Piezo는 디지털 신호 '1'에 해당하는 입력 레벨의 음성 주파수대의 펄스 신호를 입력하면 해당 주파수 소리를 출력하게 됩니다. 따라서 Piezo에 '1'의 값만 입력함으로써 Piezo를 동작 시키는 것이 아니라, 주파수를 조절하여 Piezo를 동작시킬 수 있습니다. 그러나 Piezo의 일반적인 주파수 특성에 의해 약 10KHz 이상의 주파수에서는 소리를 내지 않게 됩니다. 그리고 같은 전압 레벨에서도 주파수에 따라 소리의 강약이 달라지는 특성을 가지고 있습니다.

다음은 옥타브 및 음계별 주파수를 정리한 것입니다.

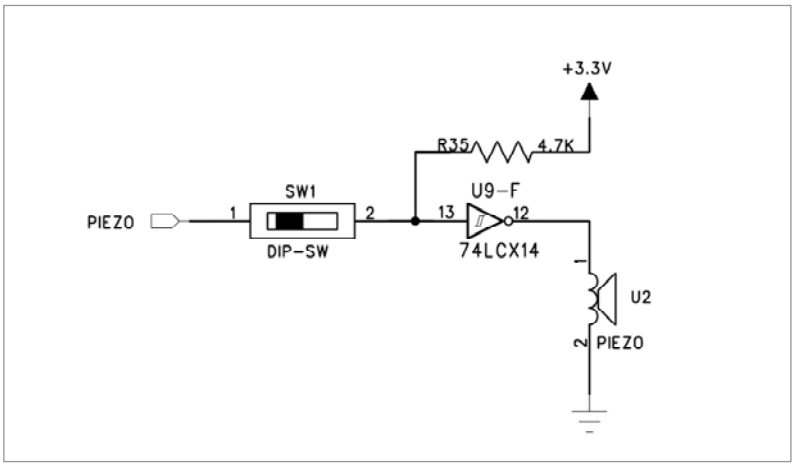
〈표 3-5〉 Piezo 옥타브 및 음계별 주파수

(단위 : Hz)

Oct	0	1	2	3	4	5	6	7
A	27.5000	55.0000	110.0000	220.0000	440.0000	880.0000	1760.000	3520.000
Bb	29.1352	58.2705	116.5409	233.0819	466.1638	932.3275	1864.655	3729.310
B	30.8677	61.7354	123.4708	246.9417	493.8833	987.7666	1975.533	3951.066
C	32.7032	65.4064	130.8128	**261.6256**	523.2511	1046.502	2093.005	4186.009
Db	34.6478	69.2957	138.5913	277.1826	554.3653	1108.731	2217.461	4434.922
D	36.7081	73.4162	146.8324	293.6648	587.3295	1174.659	2349.318	4698.636
Eb	38.8909	77.7817	155.5635	311.1270	622.2540	1244.508	2489.016	4978.032
E	41.2034	82.4069	164.8138	329.6276	659.2551	1318.510	2637.020	5274.041
F	43.6535	87.3071	174.6141	349.2282	698.4565	1396.913	2793.826	5587.652
Gb	46.2493	92.4986	184.9972	369.9944	739.9888	1479.978	2959.955	5919.911
G	48.9994	97.9989	195.9977	391.9954	783.9909	1567.982	3135.963	6271.927
Ab	51.913	103.8262	207.6523	415.3047	830.6094	1661.219	3322.438	6644.875

※ 참고: C=도, D=레, E=미, F=파, G=솔, A=라, B=시 (C음행/3옥타브열 의 261.6256Hz가 피아노의 중앙 C음에 해당함.)

3) 회로



Piezo의 회로에서는 FPGA에서 주파수 신호를 받아 소리는 내는 장치로 위 그림과 같은 회로를 구성하고 있습니다. 기본적으로는 Piezo에는 +3.3V의 High 신호가 74LCX14에 의해 반전되어 FPGA에서 아무런 신호가 없으면 Piezo에는 '0'의 신호가 기본으로 들어가게 되어 아무런 소리가 들리지 않습니다.

현재 회로에서는 딥 스위치를 사용하여 FPGA와 Piezo간의 신호를 연결 할 것인지 결정하게 됩니다. 이러한 딥 스위치의 역할은 Piezo를 구동하지 않을 때 연결 라인의 잡음에 의해 Piezo에 동작 되는 것을 방지해 주고 있습니다. 따라서 Piezo를 구동하지 않을 때는 딥 스위치를 OFF하여 잡음으로 인한 동작이 안되게 해 주어야 합니다.

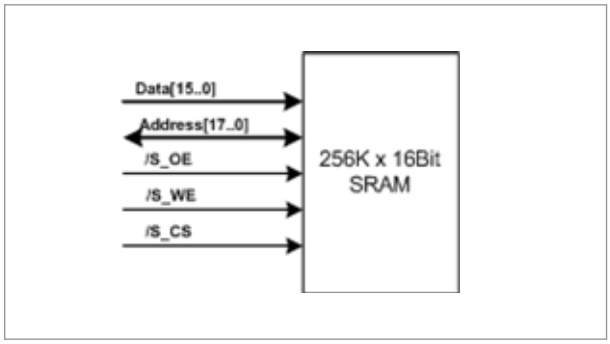
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
PIEZO	F4	H2	Speaker control

3.10 SRAM

1) 구성

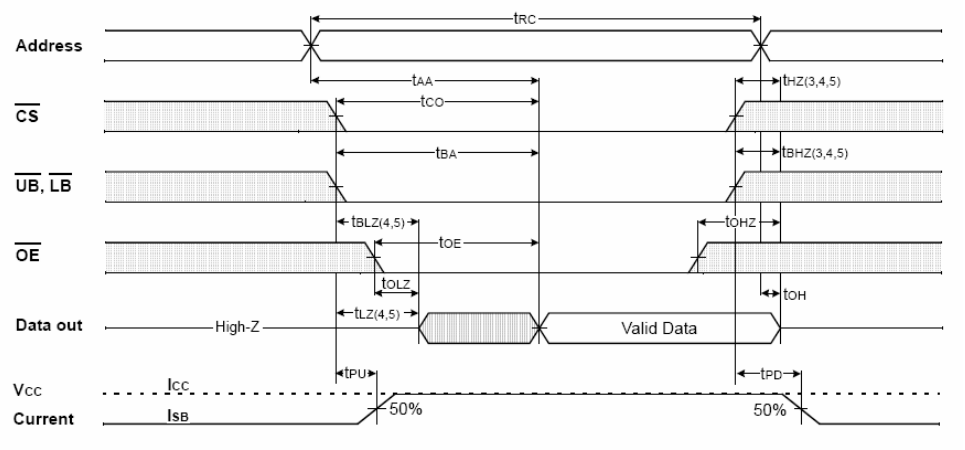
본 제품에는 디바이스 내부에서 메모리 영역으로 사용하는 공관 이외에 별도로 256Kx16 Bit, 총 4M bit의 High Speed SRAM을 기본으로 제공하고 있습니다. 이처럼 COMBO II에서는 SRAM을 통한 메모리 영역을 확장해서 사용할 수 있습니다. 그림에서 SRAM과 Flash Memory의 구성을 보여주고 있으며, [그림 3-10]에서와 같이 각각의 메모리가 독립적으로 동작할 수 있도록 각각의 라인이 분리되어 있습니다.



[그림 3-10] SRAM 블록 구성

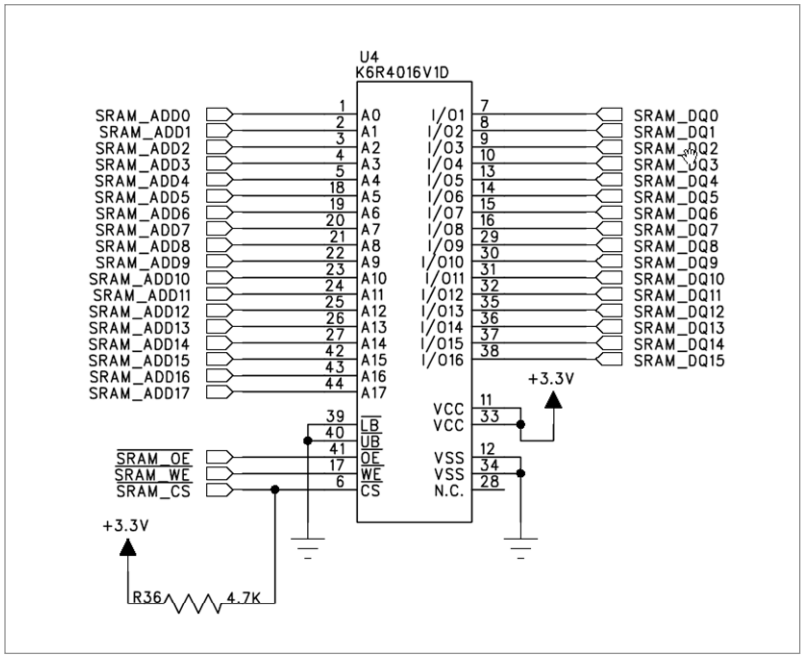
2) 동작

여기서 제공되는 SRAM (Static Random Access Memory)과 Flash Memory는 각각 FPGA I/O와 연결되어 있기 때문에 SRAM을 제어하기 위해서는 FPGA 핀 설정과 제어 회로를 설계해 주어야 합니다. 이렇게 컨트롤 핀 설정을 통하여 메모리 영역의 제어가 이루어집니다. 다음 그림에서는 SRAM에서의 READ와 WRITE에 대한 동작 사이클의 모습을 보여주고 있습니다.



[그림 3-11] SRAM read cycle

3) 회로



SRAM은 위의 그림과 같은 회로를 구성하고 있습니다. 여기에서는 18개의 Address라인과 16개의 Data 라인으로 구성되어 있고, 3개의 제어 신호를 통해 SRAM을 제어하고 있습니다. SRAM의 모든 제어 신호는 FPGA 디바이스와 직접 연결 되어 있고, FPGA 디바이스에서 SRAM 핀을 제어하여 내부에 읽고, 쓰고 하는 작업을 하게 됩니다.

4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
SRAM_ADD[0]	E1	F6	SRAM Address 0
SRAM_ADD[1]	F7	F5	SRAM Address 1
SRAM_ADD[2]	E5	G2	SRAM Address 2
SRAM_ADD[3]	E2	G1	SRAM Address 3
SRAM_ADD[4]	D2	E4	SRAM Address 4
SRAM_ADD[5]	D1	E3	SRAM Address 5
SRAM_ADD[6]	C3	D2	SRAM Address 6
SRAM_ADD[7]	C2	D1	SRAM Address 7
SRAM_ADD[8]	B5	D5	SRAM Address 8
SRAM_ADD[9]	A5	C5	SRAM Address 9
SRAM_ADD[10]	D6	A4	SRAM Address 10
SRAM_ADD[11]	C6	E5	SRAM Address 11
SRAM_ADD[12]	B6	E6	SRAM Address 12
SRAM_ADD[13]	A6	D6	SRAM Address 13
SRAM_ADD[14]	D7	B5	SRAM Address 14
SRAM_ADD[15]	C7	A5	SRAM Address 15
SRAM_ADD[16]	B7	A6	SRAM Address 16
SRAM_ADD[17]	A7	F7	SRAM Address 17
SRAM_DQ[0]	H8	C6	SRAM Data 0
SRAM_DQ[1]	E8	B6	SRAM Data 1
SRAM_DQ[2]	D8	B7	SRAM Data 2
SRAM_DQ[3]	C8	A7	SRAM Data 3
SRAM_DQ[4]	B8	E7	SRAM Data 4
SRAM_DQ[5]	A8	D7	SRAM Data 5
SRAM_DQ[6]	K9	G9	SRAM Data 6
SRAM_DQ[7]	J9	F9	SRAM Data 7
SRAM_DQ[8]	G9	B8	SRAM Data 8
SRAM_DQ[9]	F9	A8	SRAM Data 9
SRAM_DQ[10]	D9	E10	SRAM Data 10
SRAM_DQ[11]	C9	D10	SRAM Data 11
SRAM_DQ[12]	B9	G10	SRAM Data 12
SRAM_DQ[13]	A9	F10	SRAM Data 13
SRAM_DQ[14]	J10	G11	SRAM Data 14
SRAM_DQ[15]	H10	F11	SRAM Data 15
\SRAM_OE	G10	C10	Output Enable
\SRAM_WE	F10	H11	Write Enable
\SRAM_CS	E10	H12	Chip Select

3.11 Step Motor

1) 구조

COMBO II 장비에서는 모터의 구동 원리 학습을 위한 스테핑 모터 구성을 제공합니다. 이 모터는 A, B, /A, /B 4개의 데이터 라인을 가지고 모터 동작을 제어하고, 모터로 들어가는 데이터 라인의 주파수 높고 낮음에 따라서 모터의 회전 속도를 제어 할 수 있습니다.

2) 동작

모터 제어를 할 수 있도록 일반적으로 많이 사용하는 스텝 모터가 설치되어 있습니다. 위상 제어를 위해 FPGA I/O가 TR을 통해 연결되어 있으며, 모터의 제어는 여자 방식에 따라 세가지 방식으로 제어가 가능합니다. 1상 여자, 2상 여자, 1-2상 여자 방식이 있고, 이 중 1-2 여자 방식은 고-분해능의 이점을 기지고 있습니다. 다음의 표에서는 모터를 제어하기 위한 데이터의 흐름을 표로 보여주고 있습니다. 각각의 데이터 라인에 다음가 같은 표를 순차적으로 인가하여 모터를 회전 시킬 수 있습니다.

❑ 1상 여자

구분	1	2	3	4	5	6	7	8	9
A	1	0	0	0	1	0	0	0	1
B	0	1	0	0	0	1	0	0	0
/A	0	0	1	0	0	0	1	0	0
/B	0	0	0	1	0	0	0	1	0

1주기

❑ 2상 여자

구분	1	2	3	4	5	6	7	8	9
A	1	0	0	1	1	0	0	1	1
B	1	1	0	0	1	1	0	0	1
/A	0	1	1	0	0	1	1	0	0
/B	0	0	1	1	0	0	1	1	0

1주기

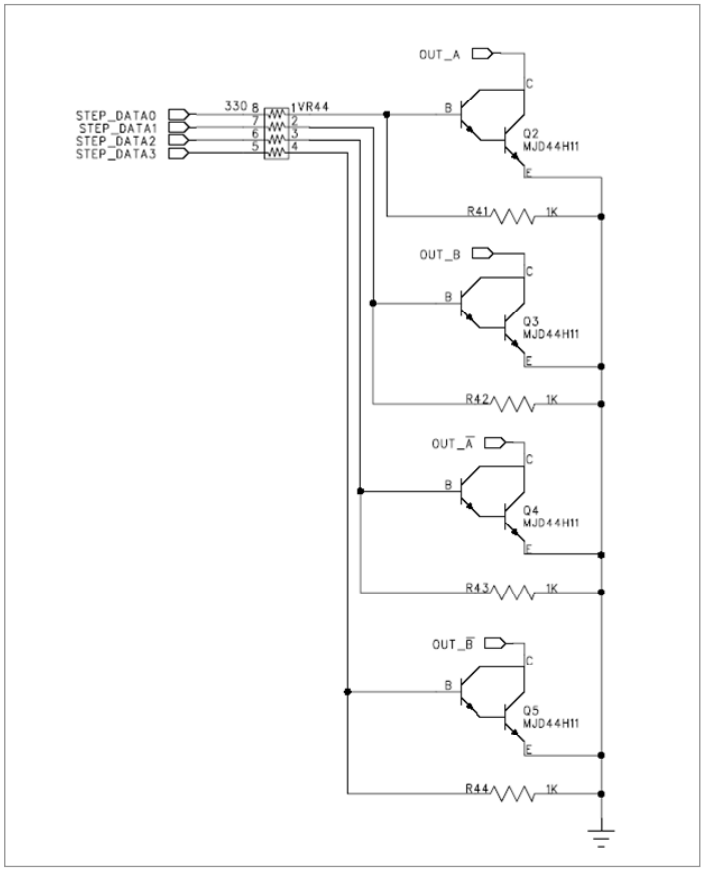
❑ 1-2상 여자

구분	1	2	3	4	5	6	7	8	9
A	1	1	0	0	0	0	0	1	1
B	0	1	1	1	0	0	0	0	0
/A	0	0	0	1	1	1	0	0	0
/B	0	0	0	0	0	1	1	1	0

1주기

FPGA에서의 모터 제어는 위 표 중 하나의 여자 방식을 선택하여 모터에 신호를 제공하여 모터를 동작하게 합니다.

3) 회로



회로에서 보는 것과 같이 A, B, /A, /B 4개의 데이터 라인이 FPGA로부터 모터로 입력되는 모습을 보여주고 있습니다. 모터에서는 FPGA로부터 출력되는 전류가 작기 때문에

중간에 NPN 트랜지스터를 사용, 전류를 증폭하여 모터로 보내게 됩니다. 따라서 모터에 사용할 수 있는 전류로 증폭이 되어 모터의 동작이 이루어 지게 됩니다.

4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
STEP_DATA[0]	AD8	Y8	Motor Data A
STEP_DATA[1]	AC8	AF7	Motor Data B
STEP_DATA[2]	AF8	AF8	Motor Data \A
STEP_DATA[3]	AE8	AE8	Motor Data \B
Motor Sensor	M5	N1	Detected High

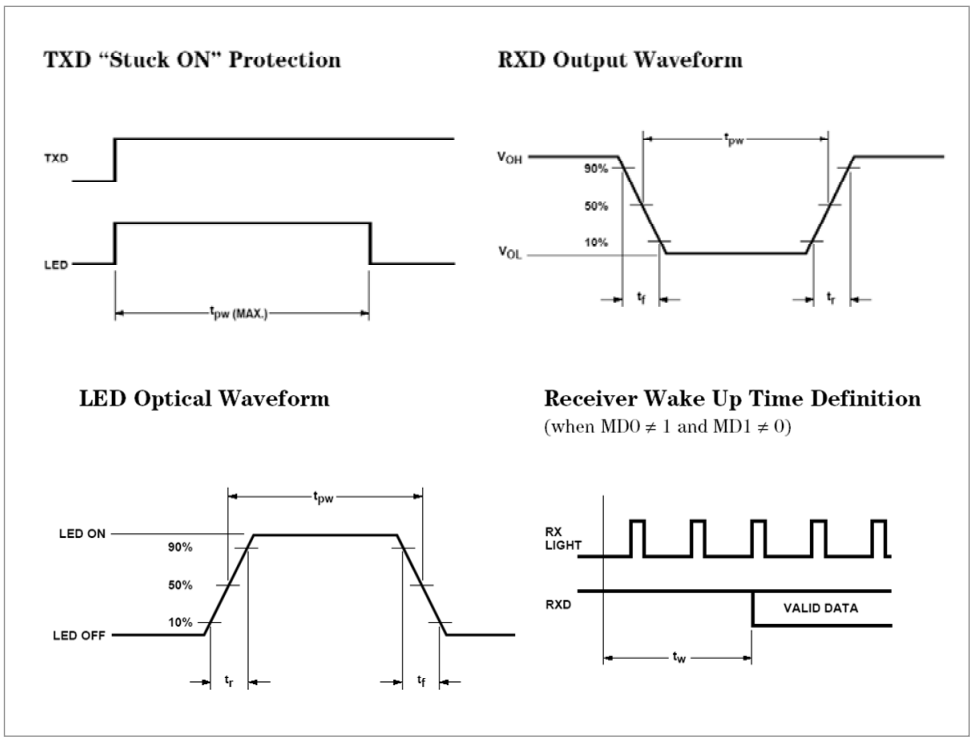
3.12 IrDA

1) 구성

COMBO II에서는 HSDL-3600시리즈의 적외선 모듈을 사용하여 IrDA 블록을 구성하고 있습니다. 이 모듈 내부에는 적외선 수신 및 발신을 하는 장치가 내부에 포함되어 있습니다. 또한 IrDA 모듈의 통신 모드를 제어하기 위해 주변 저항을 통해 모듈의 모드 설정을 해 주고 있습니다. FPGA 디바이스에서는 통신에 해당되는 데이터 선인 TX, RX에 해당되는 부분만 연결이 되어 있습니다.

2) 동작

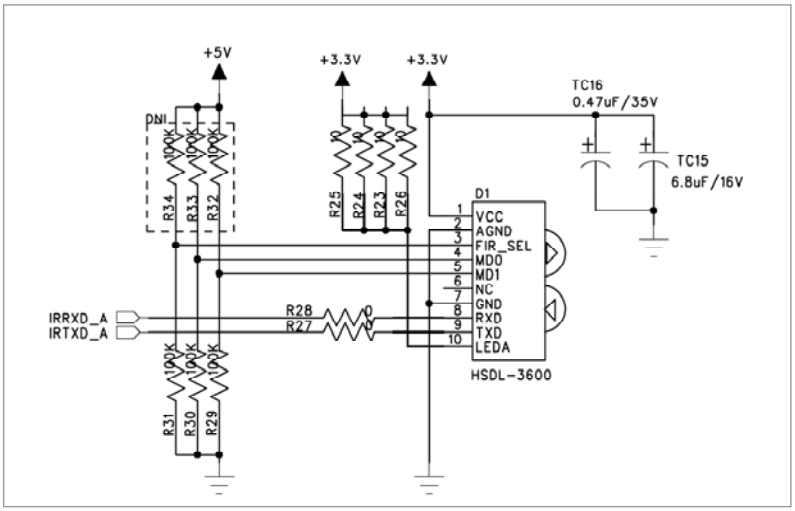
HSDL-3600모듈에서는 Mode와 FIR_SEL핀을 통해 RX, TX의 모드를 설정해 줄 수 있습니다. 아래의 표에서 핀의 설정에 따른 모드의 변화를 보여주고 있습니다. 기본 모드는 RX Function이 SIR로 되어 있고, TX Function이 Full Distance Power로 설정되어 있습니다.



〈표 3-6〉 HSDL-3600 모드

Mode 0	Mode 0	FIR_SEL	RX Function	TX Function
1	0	X	Shutdown	Shutdown
0	0	0	SIR	Full Distance Power
0	1	0	SIR	2/3 Distance Power
1	1	0	SIR	1/3 Distance Power
0	0	1	MIR/FIR	Full Distance Power
0	1	1	MIR/FIR	2/3 Distance Power
1	1	1	MIR/FIR	1/3 Distance Power

3) 회로



회로에서 보듯이 IrDA 블록에서 FPGA와 데이터가 연결되어 있는 핀은 RX, TX의 통신에 관여한 두 핀만 연결이 되어 있습니다. 나머지는 저항에 의한 풀 업, 풀 다운 모드 설정을 위한 부분과 연결이 되어 있습니다. 모드는 앞서 설명한 것과 같이 RX Function은 SIR로 되어 있고, TX Function은 Full Distance Power로 설정되어 있습니다. 따라서 사용자는 RX, TX에 대한 신호 데이터를 FPGA에서 생성하여 보내고 받는 동작에 대한 확인을 하면 됩니다.

4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
IrDA_RXD	N9	P6	Receiver Data Output
IrDA_TXD	M2	P5	Transmitter Data Input

3.13 UART

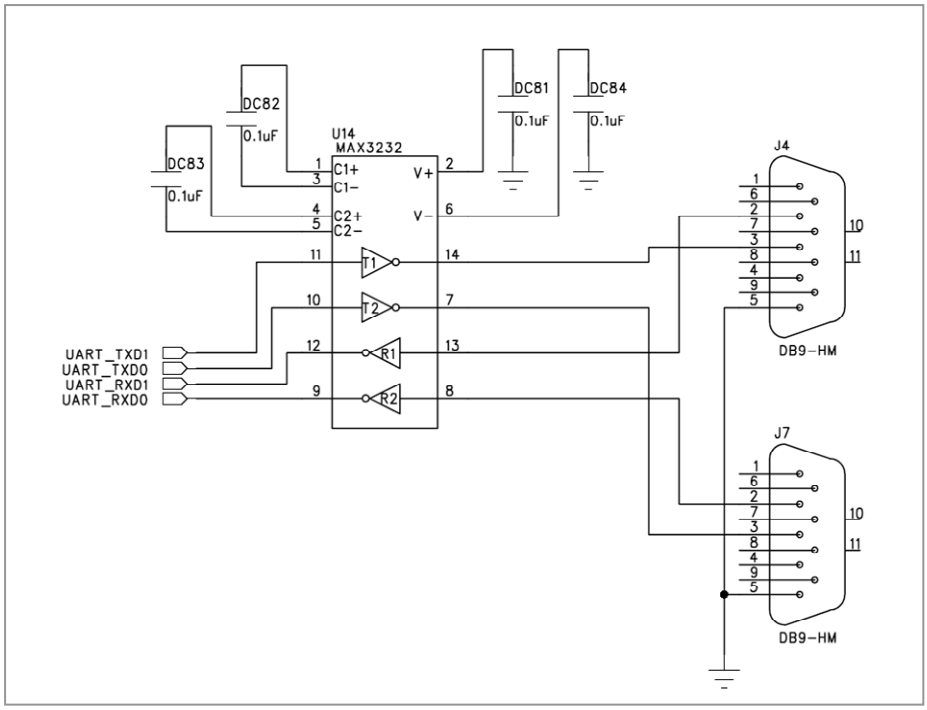
1) 구성

본 제품에는 PC의 RS232(시리얼)포트와 연결하여 시리얼 데이터 통신을 할 수 있도록 하는 Male타입의 9핀 커넥터를 2개 제공하고 있습니다. 따라서 동시에 2개의 통신 포트를 통해 상호간의 통신을 할 수 있고, 장비 내부에 있는 장치를 제어할 수 있습니다.

2) 동작

제품에서 사용된 시리얼 인터페이스용 디바이스는 Maxim사의 MAX3232계열을 사용하여 최대 1Mbps의 데이터를 송수신 할 수 있도록 구성하였으며, 데이터 포트는 RXD, TXD의 두 포트를 사용하도록 하였습니다.

3) 회로



회로에서 보는 것과 같이 MAX3232 디바이스를 사용하여 최대 1Mbps 성능의 통신 속도를 보여주고 있습니다. 또한 전압 레벨로 +3.0~+5.5V의 전압을 지원해 주고, 동시에 2개의 UART를 제어할 수 있는 디바이스입니다. 현재 2개의 시리얼 케이블을 장착할 수 있는 커넥터에 MAX3232 디바이스를 통해 FPGA 디바이스와 연결되어 있어 통신

실험을 할 수 있도록 지원하고 있습니다.

4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
UART_TXD0	L3	N7	Transmitter Data Input
UART_RXD0	L7	N5	Receiver Data Output
UART_TXD1	L6	N8	Transmitter Data Input
UART_RXD1	L9	N6	Receiver Data Output

3.14 USB/Serial 포트

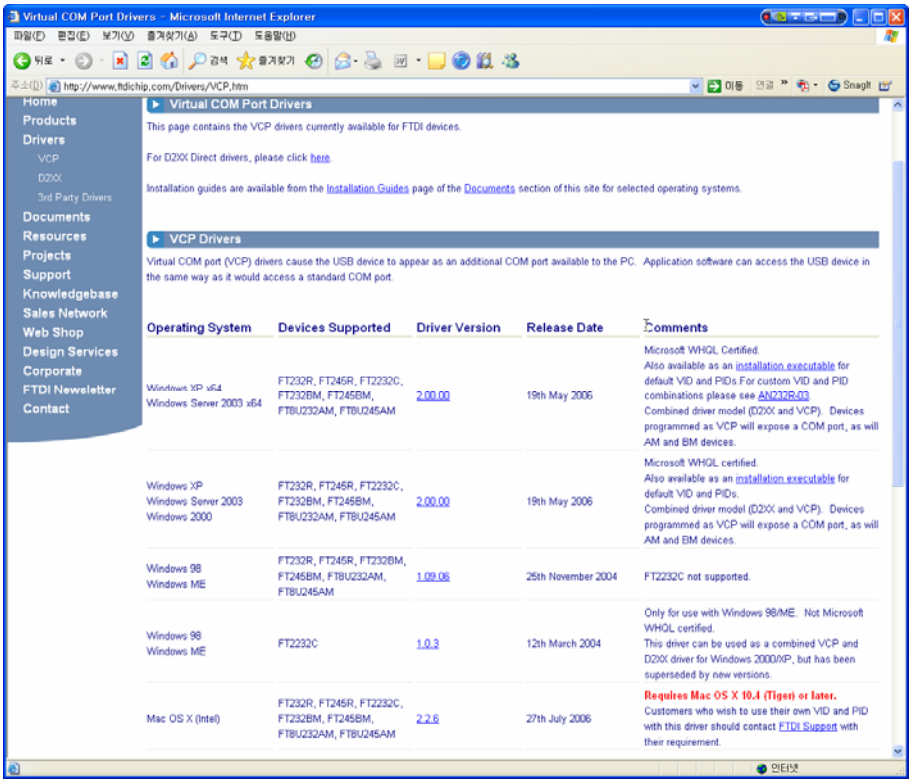
1) 구성

본 장비에서 USB 포트는 시리얼 통신을 할 수 있는 포트 입니다. 따라서 RS232 포트를 통한 통신 실험이 아닌 USB 포트를 이용한 시리얼 통신 실험을 할 수 있는 블록입니다. 이러한 구성은 현재 PC의 RS232포트가 USB 포트로 변화하는 장비에서도 시리얼 통신을 할 수 있도록 하는 USB 포트를 구성하고 있습니다.

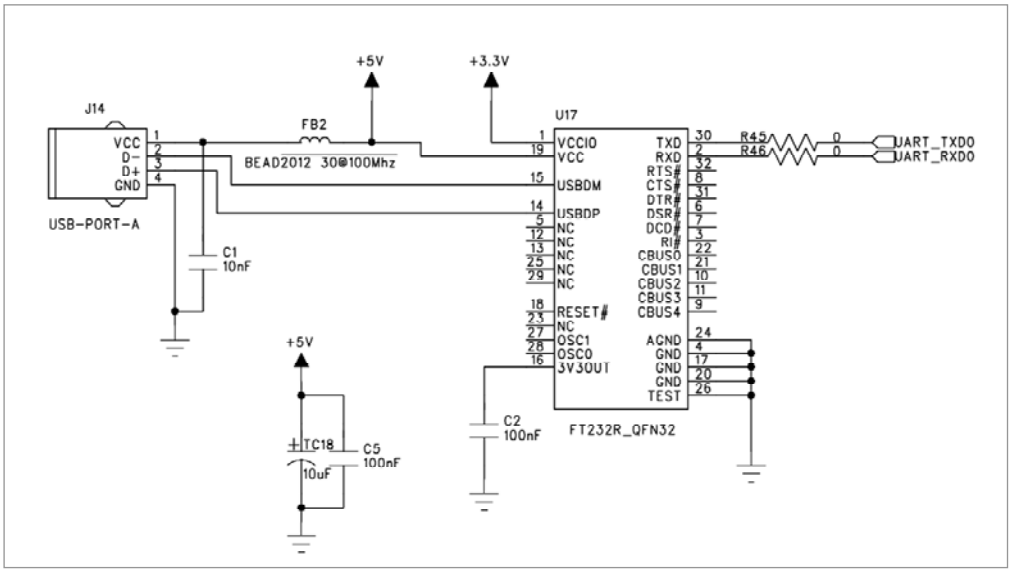
2) 동작

COMBO II에서는 FT232 디바이스를 사용하여 USB를 통한 Serial 데이터 통신 기능을 지원하고 있습니다. UART와 비슷한 핀 구조로 연결 되어 있고, FPGA 디바이스에서는 UART 2와 핀이 common으로 연결 되어 있습니다. 따라서 UART 2 와 USB는 공통의 FPGA 디바이스 핀 제어가 가능합니다.

USB 포트를 사용하기 위해서는 먼저 http://www.ftdichip.com/ 에서 FT232R에 대한 드라이버를 먼저 설치 하여야 합니다. 아래 그림에서 사이트의 다운로드 센터를 확인할 수 있습니다.



3) 회로



회로에서 보는 것과 같이 USB에서 입력되는 데이터 핀이 FT232 디바이스를 통해 바로 FPGA 디바이스로 연결되어 있습니다.

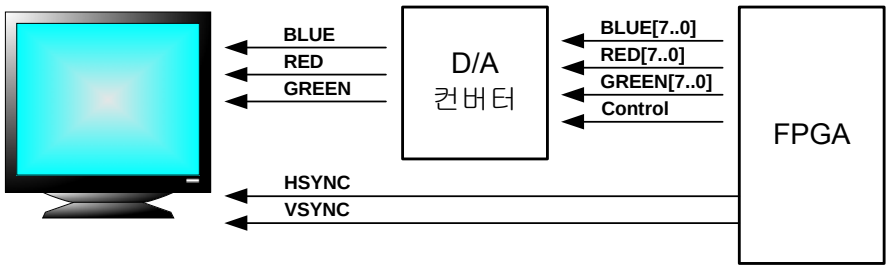
4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
USB_TXD	L3	N7	Transmitter Data Input
USB_RXD	L7	N5	Receiver Data Output

3.15 VGA

1) 구성

HBE-COMBO II는 FPGA I/O에 컬러 VGA 모니터를 구동할 수 있도록 (디지털 RGB를 아날로그 RGB 신호 변환해 줄 수 있는) D/A Converter가 연결되어 있습니다. [그림 3-12]는 데이터의 흐름에 대한 블록도를 보이고 있습니다.



[그림 3-12] VGA 블록 구성

2) 동작

VGA 모니터를 구동하기 위해 FPGA 디바이스에서 D/A 컨버터로 R (빨강), G (녹색), B (파랑) 색깔 별로 8개의 데이터 라인이 연결되어 있습니다. 이러한 D/A 컨버터를 통해 다양한 색깔을 표현할 수 있고, 1024x768 해상도에 최대 24-비트 RGB 컬러를 구현할 수 있도록 구성되어 있습니다. 사용된 D/A 컨버터의 특징은 다음과 같습니다.

- Triple 8-Bit D/A Converters
- Minimum 80 MSPS Operation
- Direct Drive of Doubly-Terminated 75-W Load Into Standard Video Levels
- 3×8 Bit 4:4:4, 2×8 Bit 4:2:2 or 1×8 Bit 4:2:2 (ITU-BT.656) Multiplexed YPbPr/GBR Input Modes
- Bi-Level (EIA) or Tri-Level (SMPTE) Sync
- Generation With 7:3 Video/Sync Ratio
- Integrated Insertion of Sync-On-Green/Luminance or Sync-On-All Channels
- Configurable Blanking Level
- Internal Voltage Reference
- High-Definition Television (HDTV) Set-Top
- Boxes/Receivers
- High-Resolution Image Processing
- Desktop Publishing

- Direct Digital Synthesis/I-Q Modulation

디바이스에 대한 보다 자세한 설명은 데이터시트를 참고로 하도록 합니다.

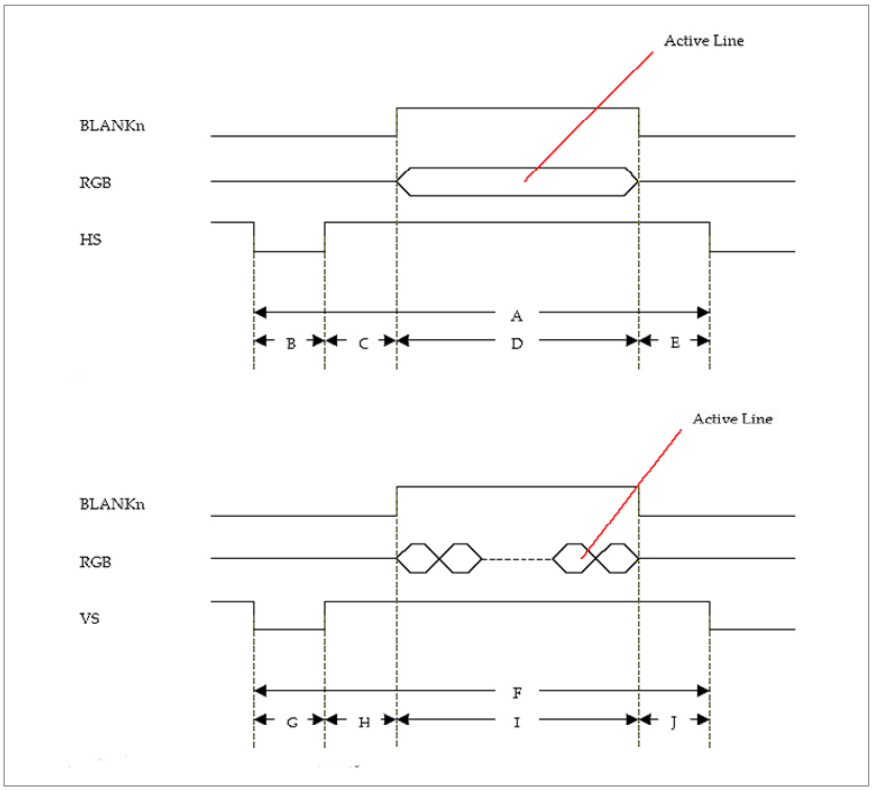
VGA 블록은 FPGA에서 VGA D/A Driver를 제어하는 회로와 VGA 모니터 동기 신호(H-sync/V-sync)를 생성해 주는 회로를 설계해 주어야 하며, 동시에 이와 관련된 I/O 핀을 적절히 지정해 주어야 합니다. 모니터 동기 신호는 구현하고자 하는 해상도에 따라 그 출력 주파수도 달라지며, 동시에 D/A 컨버터 입력 클럭도 수정되어야 합니다. 1024x768 해상도를 구현하기 위해서는 구동 클럭이 65MHz가 되어야 합니다.

다음 표는 해상도에 따른 VGA 동작 타이밍과 기타 유용한 정보를 보이고 있습니다.

〈표 3-7〉 VGA 동작 타이밍

구분	해상도	640 X 480	800 X 600	1024 X 768
A	Line Period	32.8us	26.4us	20.7us
B	Hsync Sync Period	3.8us	3.2us	2.1us
C	Hsync Back Porch	1.9us	2.2us	2.5us
D	Active Video	25.4us	20us	15.7us
E	Hsync Front Porch	0.6us	1us	0.4us
F	Frame Period	16.7ms	16.58ms	16.67ms
G	Vsync Sync Period	0.05ms	0.1ms	0.12ms
H	Vsync Back Porch	1ms	0.6ms	0.6ms
I	Active Frame	15.3ms	15.84ms	15.88ms
J	Vsync Front Porch	0.3ms	0.02ms	0.06ms

비디오 신호는 라인 수평 동기 신호와 수직 동기 신호로 구성됩니다. 해상도에 따라 수평 동기 신호는 640, 800, 1024 픽셀로 구성되며, 수직 동기 신호는 480, 600, 768 픽셀이 됩니다. Blankn 신호가 비디오 DAC 출력 (R, G, B)가 0 임을 출력합니다. 수평 동기 신호 (HS)는 새로운 라인의 시작을 의미하고, 수직 동기 신호 (VS)는 새로운 프레임의 시작을 나타냅니다.



[그림 3-13] 수평 동기 및 수직 동기

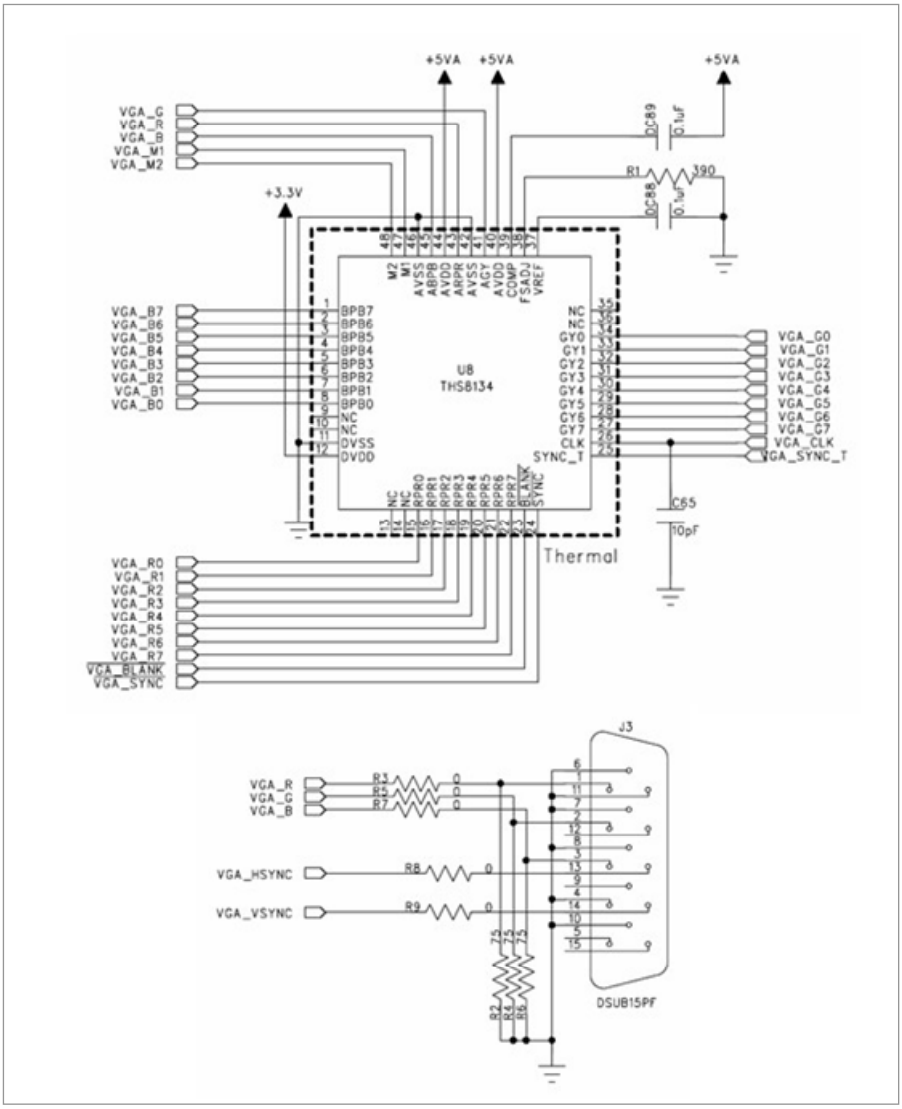
〈표 3-8〉 Video Settings

해상도	640 X 480	800 X 600	1024 X 768
Video clock	25.2MHz	40MHz	65MHz
Horizontal Resolution	640	800	1024
Vertical Resolution	480	600	768
Hsync Pulse Width	95	128	136
Hsync Back Porch width	40	88	160
Hsync Front Porch width	25	40	24
Vsync Pulse Width	2	4	6
Vsync Back Porch Width	22	23	29
Vsync Front Porch Width	10	1	3

그리고 H.Sync (수평동기)와 V.Sync (수직동기)는 각각 주사선의 수평, 수직동기를 맞추

기 위한 신호로 이 동기신호가 적절해야만 정상적인 화면을 표현할 수 있습니다. 따라서, H.Sync와 V.Sync에 적절히 동기 시켜 RGB 데이터를 입력함으로 모니터 내의 원하는 위치에 데이터를 표현할 수 있습니다.

3) 회로



회로에서 보는 것과 같이 THS8134의 D/A 컨버터를 이용하여 모니터를 제어합니다. FPGA 디바이스로부터 R (빨강), G (녹색), B (파랑) 세가지 색깔에 대한 8-bit 데이터를 THS8134 디바이스로 전달하고 이러한 데이터 값을 변환하여 모니터의 색깔을 제어하

고 있습니다.

4) 핀 구성표

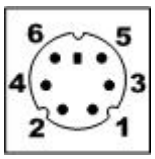
FPGA Signal	Altera	Xilinx	Description
VGA_BLUE[0]	K6	M6	Blue/Pr pixel data 0
VGA_BLUE[1]	J1	L6	Blue/Pr pixel data 1
VGA_BLUE[2]	K8	L5	Blue/Pr pixel data 2
VGA_BLUE[3]	J3	L4	Blue/Pr pixel data 3
VGA_BLUE[4]	J2	L2	Blue/Pr pixel data 4
VGA_BLUE[5]	J6	K2	Blue/Pr pixel data 5
VGA_BLUE[6]	J4	K1	Blue/Pr pixel data 6
VGA_BLUE[7]	J8	L8	Blue/Pr pixel data 7
VGA_RED[0]	J7	L7	Red/Pr pixel data 0
VGA_RED[1]	H2	K6	Red/Pr pixel data 1
VGA_RED[2]	H1	K5	Red/Pr pixel data 2
VGA_RED[3]	H4	K4	Red/Pr pixel data 3
VGA_RED[4]	H3	K3	Red/Pr pixel data 4
VGA_RED[5]	G1	J4	Red/Pr pixel data 5
VGA_RED[6]	H6	J3	Red/Pr pixel data 6
VGA_RED[7]	G3	J2	Red/Pr pixel data 7
VGA_GREEN[0]	G2	K7	Green/Y pixel data 0
VGA_GREEN[1]	G5	H1	Green/Y pixel data 1
VGA_GREEN[2]	G4	J7	Green/Y pixel data 2
VGA_GREEN[3]	F1	J6	Green/Y pixel data 3
VGA_GREEN[4]	G6	J5	Green/Y pixel data 4
VGA_GREEN[5]	F3	H5	Green/Y pixel data 5
VGA_GREEN[6]	F2	H4	Green/Y pixel data 6
VGA_GREEN[7]	F6	H3	Green/Y pixel data 7
\VGA_BLANK	K4	M8	Blanking control input
\VGA_SYNC	K7	M7	Sync control input
VGA_CLK	AA7	AD1	VGA clock
VGA_SYNC_T	K5	L1	Sync tri-level control
VGA_HSYNC	K3	M2	Horizontal Sync
VGA_VSYNC	K2	M1	Vertical Sync
VGA_M1	K1	M5	Operation mode control 1
VGA_M2	L10	M3	Operation mode control 2

3.16 PS/2 포트 사용하기

1) 구성

PS/2 포트는 주로 PC 키보드나 마우스의 입력장치로 사용되는 직렬 통신방식 입니다. 따라서 키보드 컨트롤러나 마우스 컨트롤러를 설계해 볼 수 있는 PS/2 커넥터를 제공하고 있습니다. PS/2는 데이터 선과 클럭 선 두 개만 가지고 데이터의 전송이 이루어집니다.

데이터 라인이 하나이기 때문에 데이터는 클럭에 동기 되어 시리얼 형태로 전달 됩니다. PS/2의 클럭은 키보드나 마우스 (슬레이브)가 어떤 동작 (키를 눌렀을 경우)을 할 때만 발생됩니다.



PS/2 PORT

〈표 3-9〉 PS/2 Pin Description

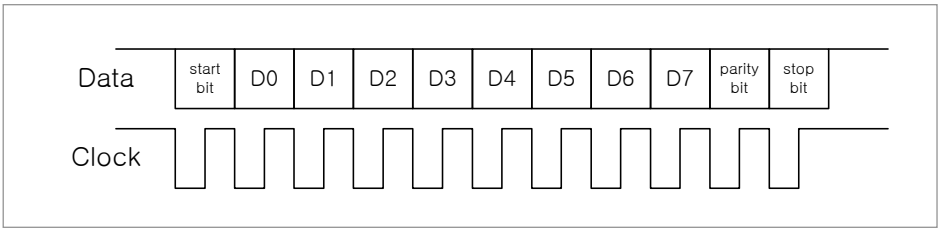
Pin	Name	Description
1	Data	Key Data
2	N/C	Not Connect
3	GND	Ground
4	VCC	+5V DC
5	CLK	Clock
6	N/C	Not Connect

데이터 형태는 시작(1-bit), 정지 비트(1-bit), 패리티 비트(1-bit), 데이터 비트(8-bit)로 구성되며, 모두 11-bit로 구성되어 있습니다. 키보드를 누르면 키보드는 해당 키의 스캔코드 값을 출력합니다.

이 코드는 처음 눌렀을 때 출력할 뿐 만 아니라 키를 뗐을 때에도 출력이 되기 때문에 키가 계속 눌려져 있을 상태를 알 수 있도록 해 줍니다. 결국 키보드는 키를 누를 때 키 값에 해당하는 마크 코드와 키를 뗐을 때 브레이크 코드의 마크 코드를 출력하므로 그 사이를 구분을 할 수 있습니다.

2) 동작

아래의 그림과 같이 키보드 통신에 사용되는 데이터 형태는 start bit, data bit 8개, parity bit, stop bit등의 총 11비트로 구성되어 있습니다. 데이터 신호는 클럭 신호로 동기를 취하는 것이 전제 되어있기 때문에 타이밍만 일치하면 데이터를 보내는 속도에 오차가 많더라도 문제가 되지 않으며 규격으로는 1비트당 60μs에서 100μs면 됩니다.



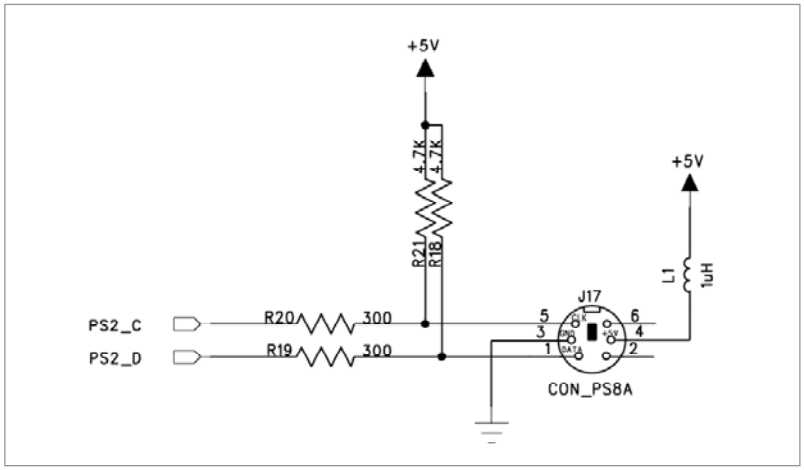
키보드의 키가 눌려지면 키보드 내부 컨트롤러에 의해 키에 대응하는 스캔 코드를 데이터 선을 통해 보냅니다. 스캔코드는 키가 떨어질 때에도 코드를 발생하기 때문에 이를 통해 키가 계속 눌러지는 것을 인식 할 수 있게 됩니다. 키를 누를 때는 make code가 떨어져 break code가 발생합니다. Make code는 1 Byte로 이루어져 있으며 break code는 make code 앞부분에 F0가 더해진 2 Byte로 이루어 집니다.

아래의 표는 해당 알파벳 및 숫자에 대한 스캔코드 입니다.

〈표 3-10〉 알파벳 숫자에 대한 스캔 코드

입력 키	Make 코드	Break 코드	입력키	Make 코드	Break 코드	입력키	Make 코드	Break 코드
1	16	F0 16	C	21	F0 21	Q	15	F0 15
2	1E	F0 1E	D	23	F0 23	R	2D	F0 2D
3	26	F0 26	E	24	F0 24	S	1B	F0 1B
4	25	F0 25	F	2B	F0 2B	T	2C	F0 2C
5	2E	F0 2E	G	34	F0 34	U	3C	F0 3C
6	36	F0 36	H	33	F0 33	V	2A	F0 2A
7	3D	F0 3D	I	43	F0 43	W	1D	F0 1D
8	3E	F0 3E	J	3B	F0 3B	X	22	F0 22
9	46	F0 46	K	42	F0 42	Y	35	F0 35
0	45	F0 45	L	4B	F0 4B	Z	1A	F0 1A
-	4E	F0 4E	M	3A	F0 3A	Enter	5A	F0 5A
=	55	F0 55	N	31	F0 31			
A	1C	F0 1C	O	44	F0 44			
B	32	F0 32	P	4D	F0 4D			

3) 회로



회로에서 보는 것과 같이 PS/2에 해당되는 클럭과 데이터 라인이 FPGA 디바이스와 연결 된 모습을 보여주고 있습니다. 이 데이터 라인으로부터 FPGA 디바이스로 스캔 코드를 읽어 들여 키보드에서 어떠한 값이 들어 왔는지 분석하여 인식하게 되는 것 입니다.

4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
PS2_CLK	M4	N4	PS/2 Clock
PS2_DATA	M3	N3	PS/2 Key data input

3.17 확장 포트

1) 구성

HBE-COMBO II 장비에서는 내부 장치 이외에 별도의 확장 포트를 사용하여 외부의 장치를 제어할 수 있도록 지원하고 있습니다. 이러한 확장 포트의 구성은 기존의 HBE-Application Module을 제어하기 위한 50핀 확장 포트 2개와 보드 내에 장착하여 사용할 수 있도록 구성한 확장 포트 하나를 가지고 있습니다. 다음은 3종류의 확장 포트에 대한 핀 구성을 보이고 있습니다.

2) 핀 구성표

❑ Expansion port #1

FPGA Signal	Altera	Xilinx	Description
EXT1[0]	J24	L26	
EXT1[1]	J25	M19	
EXT1[2]	J26	M25	
EXT1[3]	K18	M26	
EXT1[4]	K19	M22	
EXT1[5]	K21	M24	
EXT1[6]	K22	N21	
EXT1[7]	K23	N22	
EXT1[8]	K24	N19	
EXT1[9]	K25	N20	
EXT1[10]	T26	N25	
EXT1[11]	L19	N26	
EXT1[12]	L20	N23	
EXT1[13]	L21	N24	
EXT1[14]	L23	P21	
EXT1[15]	L24	P22	
EXT1[16]	L25	P19	
EXT1[17]	M19	P20	
EXT1[18]	M20	P25	
EXT1[19]	M21	P26	
EXT1[20]	M22	P23	
EXT1[21]	M23	P24	
EXT1[22]	M24	R21	
EXT1[23]	M25	R22	
EXT1[24]	N18	R19	
EXT1[25]	N20	R20	
EXT1[26]	N23	R26	

EXT1[27]	N24	T19	
EXT1[28]	P17	R24	
EXT1[29]	P18	R25	
EXT1[30]	P23	T22	
EXT1[31]	P24	T23	
EXT1[32]	R17	T20	
EXT1[33]	R19	T21	
EXT1[34]	R20	U20	
EXT1[35]	R24	U21	
EXT1[36]	R25	T25	
EXT1[37]	T17	T26	
EXT1[38]	T18	U24	
EXT1[39]	T19	U25	
EXT1[40]	T20	U22	
EXT1[41]	T21	U23	
EXT1[42]	T22	V21	
EXT1[43]	T23	V22	
EXT1[44]	T24	U26	
EXT1[45]	T25	V20	

❑ Expansion port #2

FPGA Signal	Altera	Xilinx	Description
EXT2[0]	U20	V25	
EXT2[1]	U21	W22	
EXT2[2]	U22	V23	
EXT2[3]	U23	V24	
EXT2[4]	U24	W25	
EXT2[5]	U25	W26	
EXT2[6]	U26	W23	
EXT2[7]	V20	W24	
EXT2[8]	V21	AC25	
EXT2[9]	V22	AC26	
EXT2[10]	V23	Y25	
EXT2[11]	V24	Y26	
EXT2[12]	Y18	AE24	
EXT2[13]	V18	AF23	
EXT2[14]	AA18	AF24	
EXT2[15]	AE17	AF22	
EXT2[16]	AD17	AE22	
EXT2[17]	U18	AE23	

EXT2[18]	AF17	AD23	
EXT2[19]	W17	AB22	
EXT2[20]	V17	AF21	
EXT2[21]	AC17	AD22	
EXT2[22]	AA17	AC22	
EXT2[23]	AD16	AC21	
EXT2[24]	AC16	AB21	
EXT2[25]	U17	AE21	
EXT2[26]	AE16	AD21	
EXT2[27]	W16	AC20	
EXT2[28]	AE15	AB20	
EXT2[29]	AA16	AF20	
EXT2[30]	Y16	AE20	
EXT2[31]	AB15	AE19	
EXT2[32]	AA15	AA18	
EXT2[33]	AD15	AA20	
EXT2[34]	AC15	AF19	
EXT2[35]	AC14	AC17	
EXT2[36]	AA14	AB17	
EXT2[37]	Y15	Y18	
EXT2[38]	W15	AD17	
EXT2[39]	AF13	AC16	
EXT2[40]	AE13	AB16	
EXT2[41]	Y14	AA17	
EXT2[42]	V14	Y17	
EXT2[43]	V13	W16	
EXT2[44]	AE12	AF15	
EXT2[45]	AA13	AA16	

❑ Expansion port_Daughter

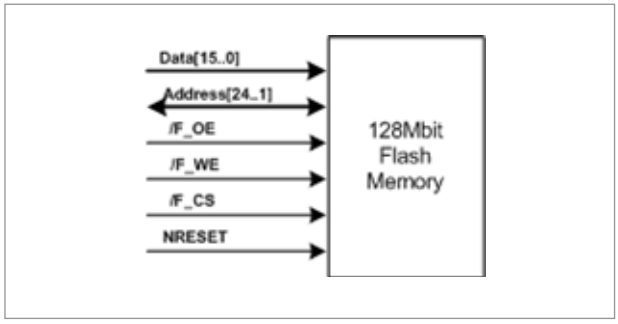
FPGA Signal	Altera	Xilinx	Description
EXT_DAU[0]	H21	L20	
EXT_DAU[1]	H19	L19	
EXT_DAU[2]	G24	K22	
EXT_DAU[3]	G23	K21	
EXT_DAU[4]	G26	K24	
EXT_DAU[5]	G25	K23	
EXT_DAU[6]	F26	J24	
EXT_DAU[7]	F25	J23	
EXT_DAU[8]	G22	K20	

EXT_DAU[9]	G21	J25	
EXT_DAU[10]	F21	J20	
EXT_DAU[11]	F20	H26	
EXT_DAU[12]	F24	J22	
EXT_DAU[13]	F23	J21	
EXT_DAU[14]	E24	H23	
EXT_DAU[15]	E23	H22	
EXT_DAU[16]	E26	H25	
EXT_DAU[17]	E25	H24	
EXT_DAU[18]	D25	E24	
EXT_DAU[19]	D23	E23	
EXT_DAU[20]	E22	H21	
EXT_DAU[21]	D26	H20	
EXT_DAU[22]	B25	D26	
EXT_DAU[23]	B24	D25	
EXT_DAU[24]	C25	C26	
EXT_DAU[25]	C24	C25	
EXT_DAU[26]	B19	B23	
EXT_DAU[27]	C19	C23	
EXT_DAU[28]	A19	A23	
EXT_DAU[29]	B18	C22	
EXT_DAU[30]	D18	D22	
EXT_DAU[31]	D19	A22	
EXT_DAU[32]	A18	B22	
EXT_DAU[33]	G18	B21	
EXT_DAU[34]	J18	C21	
EXT_DAU[35]	E18	E22	
EXT_DAU[36]	F18	A21	
EXT_DAU[37]	C17	F21	
EXT_DAU[38]	D17	A20	
EXT_DAU[39]	A17	D21	
EXT_DAU[40]	B17	E21	
EXT_DAU[41]	H17	E20	
EXT_DAU[42]	J17	F20	
EXT_DAU[43]	F17	B20	

3.18 Flash Memory (Option)

1) 구성

본 장비에서는 휘발성을 가진 SRAM과는 달리 비 휘발성의 Flash memory를 사용할 수 있도록 구성하고 있습니다. Flash Memory는 128Mbit의 용량을 가진 Intel Embedded Flash Memory를 사용하고 있습니다. 이 장치는 Option으로 기본으로 나가는 장비에서는 제공을 하지 않고 있습니다.



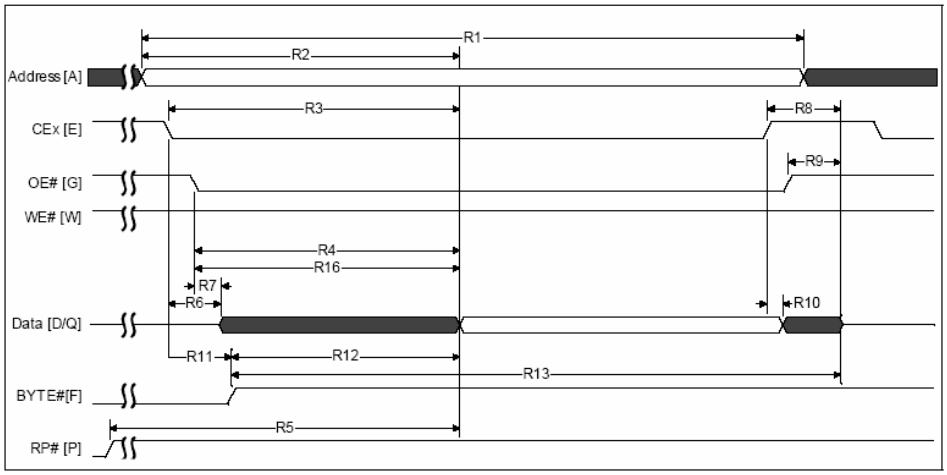
[그림 3-14] Flash Memory 구성

2) 동작

Flash Memory의 동작은 Read, Write, Reset로 나뉩니다. 다음 그림에서 각각의 동작 파형을 보이고 있습니다.

❑ Read

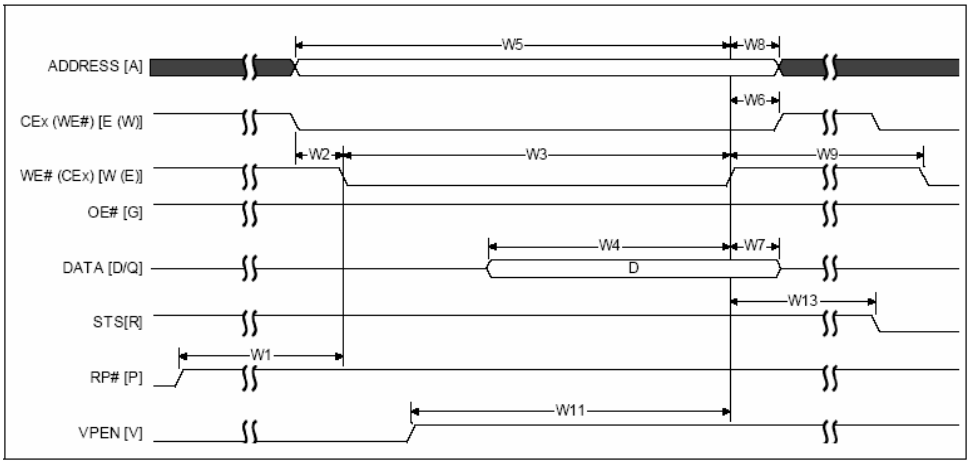
#	Sym	Parameter	Density	-75		-95		Unit	Notes
				Min	Max	Min	Max		
R1	t_{AVAV}	Read/Write Cycle Time	32 Mbit	75				ns	1,2
			64 Mbit	75					1,2
			128 Mbit	75					1,2
R2	t_{AVQV}	Address to Output Delay	32 Mbit		75			ns	1,2
			64 Mbit		75				1,2
			128 Mbit		75				1,2
R3	t_{ELQV}	CEx to Output Delay	32 Mbit		75			ns	1,2
			64 Mbit		75				1,2
			128 Mbit		75				1,2
R4	t_{GLQV}	OE# to Non-Array Output Delay			25		25	ns	1,2,4
R5	t_{PHQV}	RP# High to Output Delay	32 Mbit		150			ns	1,2
			64 Mbit		180				1,2
			128 Mbit		210				1,2
R6	t_{ELQX}	CEx to Output in Low Z	All	0		0		ns	1,2,5
R7	t_{GLQX}	OE# to Output in Low Z		0		0		ns	1,2,5
R8	t_{EHQZ}	CEx High to Output in High Z			25		25	ns	1,2,5
R9	t_{GHQZ}	OE# High to Output in High Z			15		15	ns	1,2,5
R10	t_{OH}	Output Hold from Address, CEx, or OE# Change, Whichever Occurs First		0		0		ns	1,2,5
R11	t_{ELFL}/ t_{ELFH}	CEx Low to BYTE# High or Low			10		10	ns	1,2,5
R12	t_{FLQV}/ t_{FHQV}	BYTE# to Output Delay			1		1	µs	1,2
R13	t_{FLQZ}	BYTE# to Output in High Z			1		1	µs	1,2,5
R14	t_{EHEL}	CEx High to CEx Low	All	0		0		ns	1,2,5
R15	t_{APA}	Page Address Access Time			25		25	ns	5, 6
R16	t_{GLQV}	OE# to Array Output Delay							



[그림 3-15] Read Operations

❑ Write

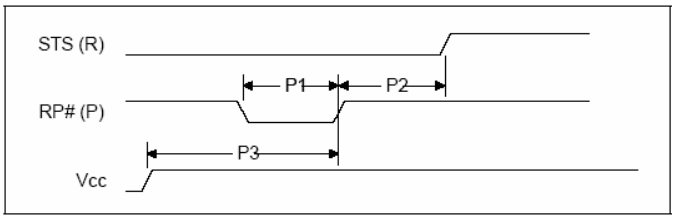
#	Symbol	Parameter	Density	Valid for All Speeds		Unit	Notes
				Min	Max		
W1	t_{PHWL} (t_{PHEL})	RP# High Recovery to WE# (CE_X) Going Low	32 Mbit	150		ns	1,2,3
			64 Mbit	180			
			128 Mbit	210			
W2	t_{ELWL} (t_{WLEL})	CE_X (WE#) Low to WE# (CE_X) Going Low	All	0			1,2,4
W3	t_{WP}	Write Pulse Width		60			1,2,4
W4	t_{DVWH} (t_{DVEH})	Data Setup to WE# (CE_X) Going High		50			1,2,5
W5	t_{AVWH} (t_{AVEH})	Address Setup to WE# (CE_X) Going High		55			1,2,5
W6	t_{WHEH} (t_{EHWH})	CE_X (WE#) Hold from WE# (CE_X) High		0			1,2,
W7	t_{WHDX} (t_{EHDX})	Data Hold from WE# (CE_X) High		0			1,2,
W8	t_{WHAX} (t_{EHAX})	Address Hold from WE# (CE_X) High		0			1,2,
W9	t_{WPH}	Write Pulse Width High		30			1,2,6
W11	t_{VPWH} (t_{VPEH})	V_{PEN} Setup to WE# (CE_X) Going High		0			1,2,3
W12	t_{WHGL} (t_{EHGL})	Write Recovery before Read		35			1,2,7
W13	t_{WHRL} (t_{EHRL})	WE# (CE_X) High to STS Going Low			500		1,2,8
W15	t_{QVVL}	V_{PEN} Hold from Valid SRD, STS Going High		0			1,2,3,8,9



[그림 3-16] Write Operations

❑ Reset

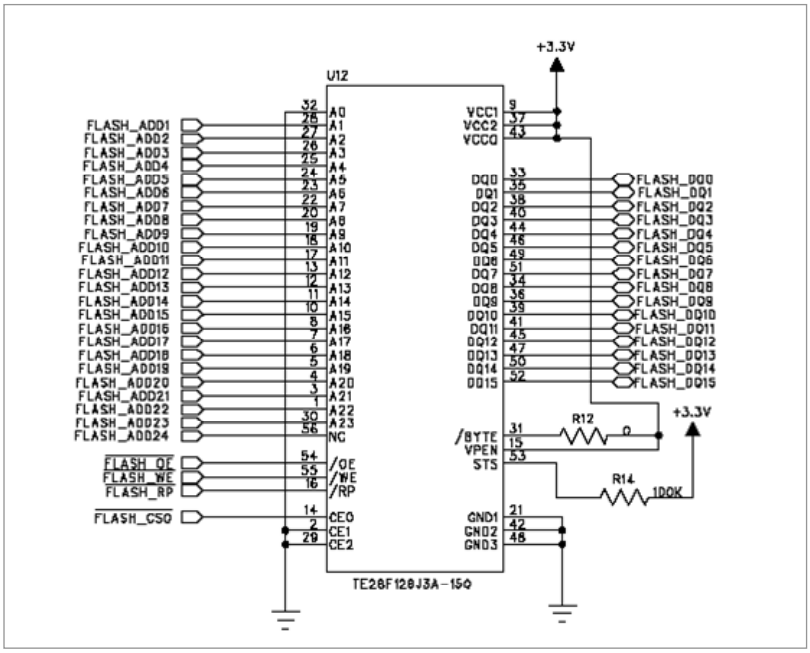
#	Symbol	Parameter	Min	Max	Unit	Notes
P1	t_{PLPH}	RP# Pulse Low Time (If RP# is tied to V_{CC}, this specification is not applicable)	25		µs	1,2
P2	t_{PHRH}	RP# High to Reset during Block Erase, Program, or Lock-Bit Configuration		100	ns	1,3
P3	t_{VCCPH}	Vcc Power Valid to RP# de-assertion (high)	60		µs	



[그림 3-17] Reset Operation

3) 회로

Flash Memory의 회로에서 알 수 있듯이 Data와 Address 라인이 FPGA 디바이스로 연결되어 있습니다. 또한 Flash Memory의 Control 라인도 연결이 되어 있어 FPGA 디바이스에서 Flash Memory의 제어가 가능하게 회로가 구성되어 있습니다.



4) 핀 구성표

FPGA Signal	Altera	Xilinx	Description
Flash_ADD[1]	D10	G12	Flash Address 1
Flash_ADD[2]	C10	E11	Flash Address 2
Flash_ADD[3]	B10	D11	Flash Address 3
Flash_ADD[4]	A10	C12	Flash Address 4
Flash_ADD[5]	J11	B12	Flash Address 5

Flash_ADD[6]	H11	F12	Flash Address 6
Flash_ADD[7]	G11	E12	Flash Address 7
Flash_ADD[8]	F11	G13	Flash Address 8
Flash_ADD[9]	D11	F13	Flash Address 9
Flash_ADD[10]	C11	A12	Flash Address 10
Flash_ADD[11]	B11	H13	Flash Address 11
Flash_ADD[12]	H12	C13	Flash Address 12
Flash_ADD[13]	G12	H14	Flash Address 13
Flash_ADD[14]	F12	E13	Flash Address 14
Flash_ADD[15]	E12	D13	Flash Address 15
Flash_ADD[16]	D12	E14	Flash Address 16
Flash_ADD[17]	C12	D14	Flash Address 17
Flash_ADD[18]	B12	G14	Flash Address 18
Flash_ADD[19]	J13	F14	Flash Address 19
Flash_ADD[20]	G13	G15	Flash Address 20
Flash_ADD[21]	F13	F15	Flash Address 21
Flash_ADD[22]	J14	A14	Flash Address 22
Flash_ADD[23]	G14	H15	Flash Address 23
Flash_ADD[24]	F14	B15	Flash Address 24
Flash_DQ[0]	D14	A15	Flash Data 0
Flash_DQ[1]	B14	E15	Flash Data 1
Flash_DQ[2]	A14	C15	Flash Data 2
Flash_DQ[3]	H15	F16	Flash Data 3
Flash_DQ[4]	G15	E16	Flash Data 4
Flash_DQ[5]	F15	H16	Flash Data 5
Flash_DQ[6]	E15	G16	Flash Data 6
Flash_DQ[7]	D15	F17	Flash Data 7
Flash_DQ[8]	C15	E17	Flash Data 8
Flash_DQ[9]	B15	D16	Flash Data 9
Flash_DQ[10]	K16	G17	Flash Data 10
Flash_DQ[11]	J16	G18	Flash Data 11
Flash_DQ[12]	H16	F18	Flash Data 12
Flash_DQ[13]	G16	D17	Flash Data 13
Flash_DQ[14]	F16	C17	Flash Data 14
Flash_DQ[15]	D16	B19	Flash Data 15
\Flash_OE	C16	A19	Output Enable
\Flash_WE	B16	E18	Write Enable
\Flash_RP	K17	G19	RESET
\Flash_CE	J17	D20	Chip Enable

04

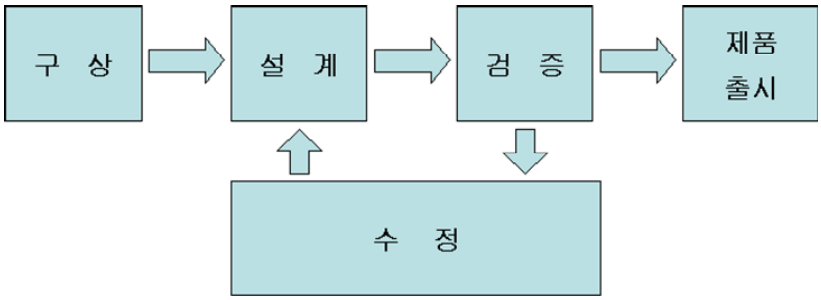
HBE-COMBO II
User's Manual &
Lab Guide

설계 S/W 기본사용법

4. 설계S/W 기본사용법

4.1 Design Flow

일반적으로 어떤 제품을 개발할 때, 아래 그림과 같은 과정으로 진행합니다.

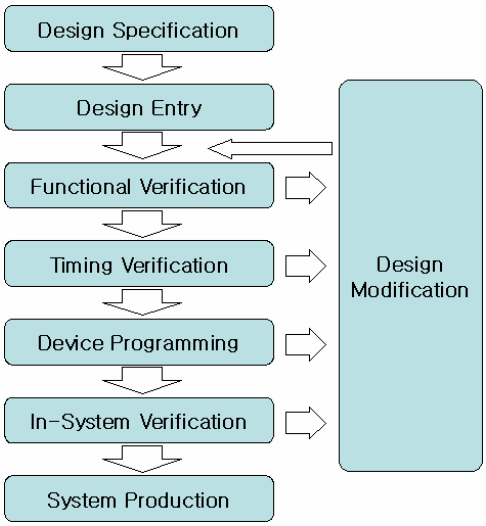


[그림 4-1] 제품 생산의 흐름

어떤 제품을 만들 것인지를 생각하고 (구상), 그 물건을 만들고 (설계), 그 물건이 의도한 대로 만들어 졌는지 테스트 (검증)합니다. 만약에 이상이 발견되면 다시 수정하여 검증하고, 이상이 없다면 제품을 출시하는 과정을 거칩니다.

하나의 시스템을 설계 하는 것도 마찬가지인데, [그림 4-2]는 설계 소프트웨어를 이용해서 시스템을 설계하는 과정입니다.

먼저 Design Specification에서는 어떤 논리 회로를 설계할 것인가를 구상합니다. 구체적으로 어떤 입력을 받아 어떻게 동작하여 어떤 핀으로 출력을 할 것 인지와 같은 내용을 구상하는 것입니다.



[그림 4-2] 시스템 설계 시 설계 흐름

그리고, Design Entry에서 설계하고자 하는 논리 회로를 어떤 방법으로 어떻게 설계할 것인가를 결정하여 논리 회로를 설계합니다. 여기에서 어떤 방법에 해당하는 논리 회로를 설계하는 방법은 여러 가지 있는데, Graphic Design (Schematic Capture에 의한 방법)으로 설계하는 방법, HDL (Hardware Description Language: 하드웨어 기술 언어)를 사용하여 설계하는 방법, 파형으로 설계하는 방법이 그것입니다.

위에서 설계한 내용에 대해 문법을 검사하여 이상이 있다면 그 이상에 대한 메시지 등을 알려주고, 이상이 없다면 설계한 논리 회로를 하드웨어적으로 만들거나 시뮬레이션 할 수 있는 파일 또는 프로그래밍 할 수 있는 파일 등을 생성하는 부분이 Compilation입니다.

이렇게 문법 오류까지 검증된 논리 회로는 설계한 의도대로 동작이 잘 되는지에 대한 검증 과정을 거치게 됩니다. 먼저 기능적으로 문제가 없는지를 살펴보는 Function Verification, 그 후에 직접 디바이스의 특성을 이용하여 설계한 회로가 사용하고자 하는 디바이스나 클럭에 따른 오동작을 하지 않는지를 살펴보는 Timing Verification 과정을 진행합니다.

위의 Function Verification과 Timing Verification은 PC상에서 소프트웨어로 검증하는 방법이고, 이렇게 검증된 논리 회로가 직접 하드웨어 (시스템)에 연결되었을 때 정확하게 동작이 되는지를 검증해야 할 필요가 있습니다. 이 부분이 Device Programming과 In-System Verification입니다. 전자는 말 그대로 디바이스에 설계한 논리 회로를 프로그래밍 한다는 말이고, 후자는 시스템이 꾸며진 상태에서 하드웨어 테스트를 한다는 것입니다.

이렇게 모든 검증이 끝난 후에 비로서 제품이 완성되어 제조됩니다.

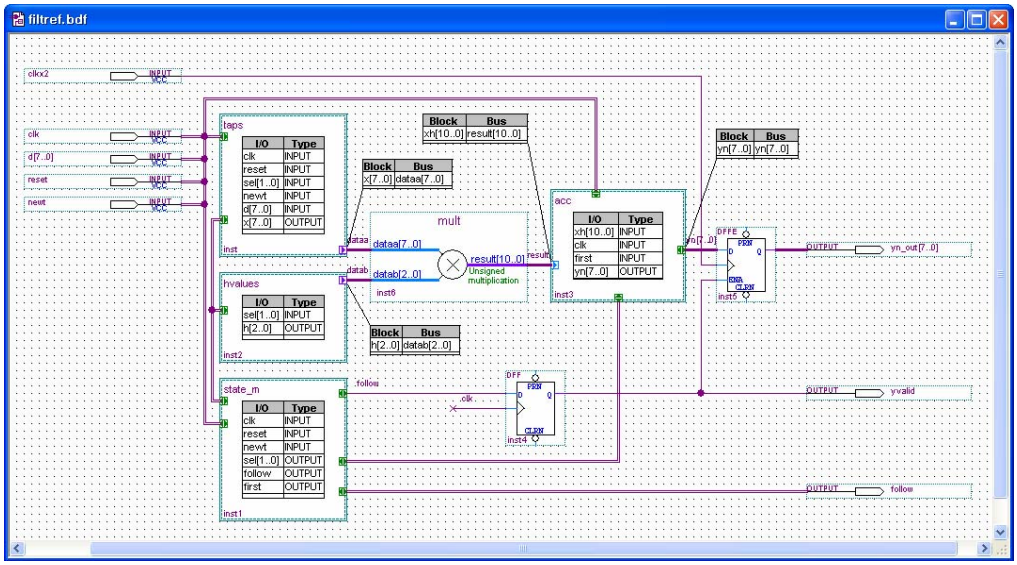
4.2 설계 소프트웨어

4.2.1 Quartus II를 이용한 설계

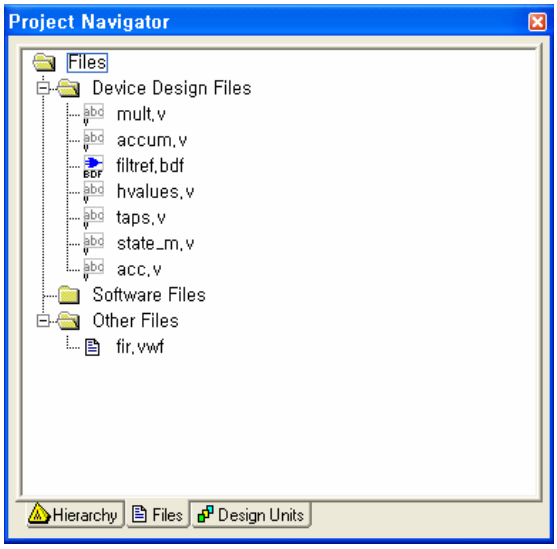
1) 역할

Altera Quartus II 프로그램은 모든 설계를 프로젝트 중심으로 관리합니다. 프로젝트에 포함된 설계 파일 및 컴파일 한 파일, 환경저장 파일, 시뮬레이션 파일, 프로그래밍 파일이 모두 같은 프로젝트 이름으로 관리가 됩니다.

또한 모든 설계를 프로젝트화 하였기 때문에 전체 프로젝트에 대한 관리도 각각의 프로젝트 별로 관리하게 되어 있습니다. [그림 4-3]은 Quartus II에서 기본으로 제공해 주는 예제 중 Fir-filter라는 예제이고, [그림 4-4]는 이 Fir-filter 프로젝트에 대한 Hierarchy Display로, Fir-filter에 대한 하위 블록들을 계층구조로 보여주는 것입니다. [그림 4-4]와 같이 프로젝트에 포함된 설계 파일 및 하위 파일들은 쉽게 파악할 수 있습니다. 또한 이러한 파일들은 Setting 메뉴를 통해 파일 뿐만 아니라 프로젝트에 대한 전반적인 관리를 쉽게 하도록 구성되어 있습니다.



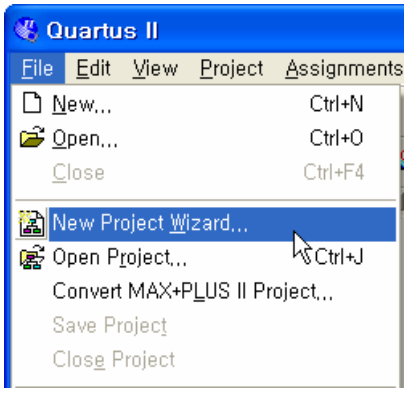
[그림 4-3] Fir-filter Project



[그림 4-4] Hierarchy Display

2) Project 선언

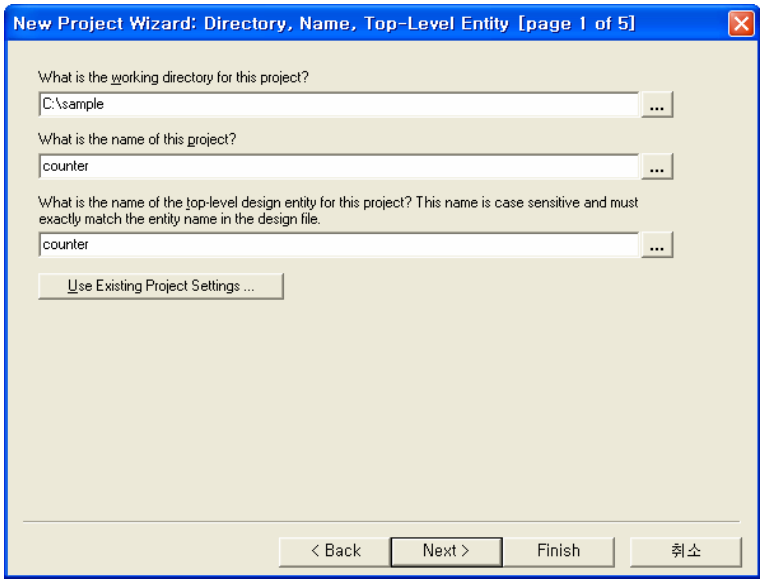
[그림 4-5]와 같이 Quartus II 프로그램은 File -> New Project Wizard 메뉴를 선택해서 현재 작업하는 프로젝트를 선언할 수 있습니다. 프로젝트 선언은 프로젝트 명 선언에서부터 디바이스 선택 등의 총 6단계로 이루어져 있는데, 이 과정을 거쳐야 실제 코딩을 위한 프로젝트 선언 과정이 완료됩니다.



[그림 4-5] New Project Wizard

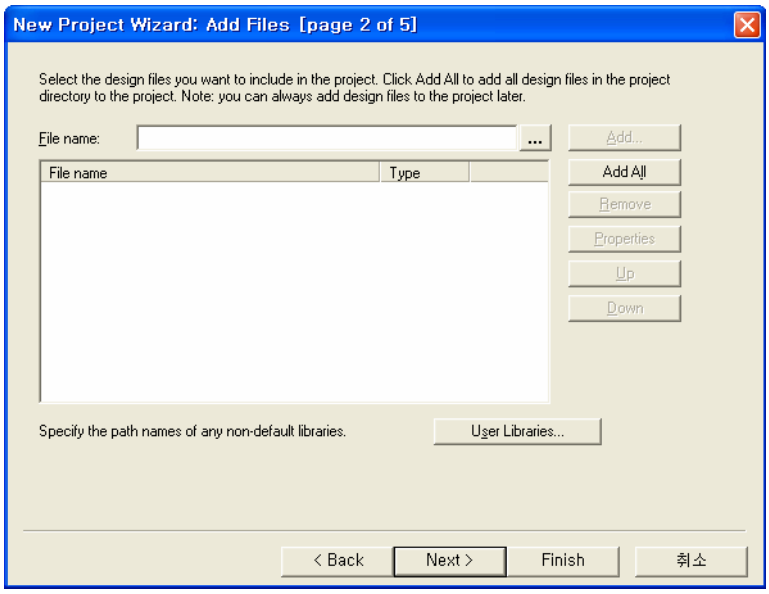
각 단계에서 다음 단계로 진행하기 위해서는 Next 버튼을 클릭해야 합니다. 프로젝트 선언을 시작하면 프로젝트에 대한 전반적인 진행 사항을 설명하는 창이 활성화되고 Next 버튼에 의해 다음 프로젝트 선언을 시작하게 됩니다.

첫 단계로 프로젝트 관련 파일들이 저장될 Directory, Project, Top-Level의 Entity 이름을 설정 합니다. 이때 주의할 사항은 VHDL등의 Language로 작성할 경우 코드의 Entity 이름과 현재 저장하는 Top-Level의 Entity의 이름을 같게 해 주어야 합니다. [그림 4-6]에서는 프로젝트의 이름과 Entity의 이름을 동일하게 counter로 설정해 주었고, 이에 관련된 파일들이 C:\sample에 저장되도록 설정했습니다.



[그림 4-6] Directory, Name, Top-Level Entity 설정

다음으로 다른 프로젝트 파일에서 필요한 파일들을 추가하거나, 그 프로젝트의 모든 파일들을 참조하기 위한 Library Path를 설정해 주는 부분입니다. 이 부분은 여러 사람들이 함께 작업하는 설계과정에서 유용하게 사용됩니다. 따라서 여러 사람들이 각각의 프로젝트를 나누어 설계하고, 최종 top 프로젝트에서 하나로 모아 설계합니다. [그림 4-7]은 파일들을 Add 버튼에 의해서 필요한 파일들을 프로젝트에 추가하게 됩니다.



[그림 4-7] Add Files

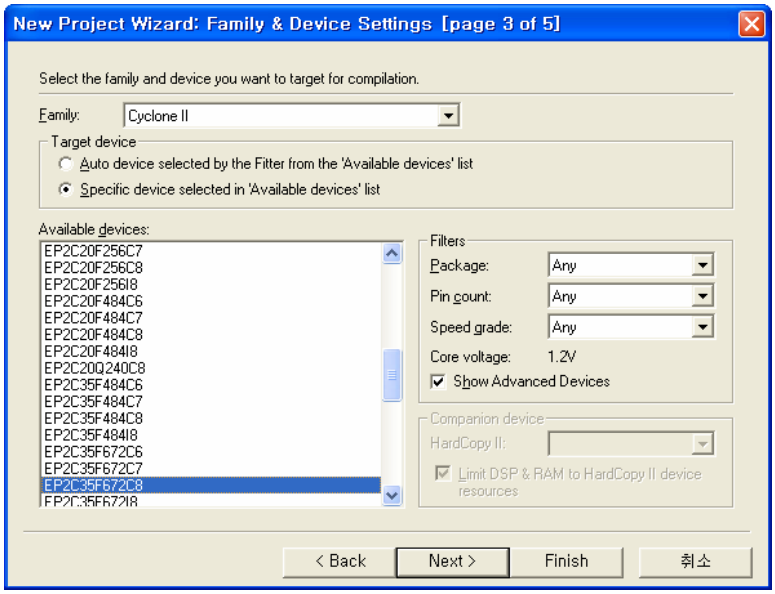
다음으로 Family & Device Settings 설정 부분입니다. Device Family는 사용하는 디바이스가 RAM 또는 ROM 타입이냐에 따라 FPGA, CPLD로 구분 지을 수 있고, 세부적으로 Device 용량 및 디바이스가 가지고 있는 핀 수, 처리 속도에 따라 나누어 집니다. 종류를 보면, Stratix II, Stratix, Cyclone II, Cyclone, APEX II, APEX 20K, & FLEX 10K의 수백만 게이트 단계까지 설계가 가능하고, SoC라는 하나의 디바이스에서 모든 구현을 해 볼 수 있는 제품도 있습니다.

또한 이러한 디바이스에서 주로 사용하는 기능만으로 구성되어 있는 Cyclone II, Cyclone, APEX 1K 가 있습니다. 또한 고속 통신이 가능한 Stratix GX, Mercury가 있고, CPLD로 MAX Series가 있습니다. 이렇게 다양한 Device Family를 Tool에서 선택할 수 있습니다. [그림 4-8]에서는 현재 COMBO II 장비에서 사용하는 Family 군인 cyclone II를 선택한 것을 볼 수 있습니다.

그리고 밑에 보이는 Target device에서의 체크하는 부분은 현재 결정한 Device Family 내에 있는 여러 디바이스 중 타겟 디바이스를 선택해서 프로젝트에 적용할 것인지, 아니면 컴파일러가 디바이스 중 적당한 디바이스를 자동으로 선택해서 프로젝트에 적용할 것인지를 선택하는 작업입니다.

예를 들어 사용자가 타겟 디바이스를 가지고 설계를 할 때, Target device는 “Specific device selected in ‘Available device’ list”에 체크를 해 주고 가지고 있는 디바이스를 선택해 주면 됩니다. 또한 오른 쪽 화면에 있는 Filters 부분은 Family 내에 있는 여러 디바이

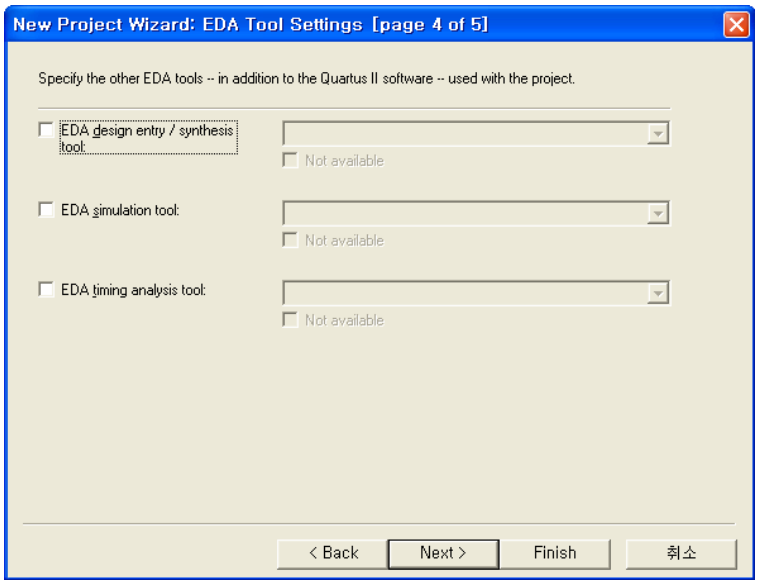
스 중, 디바이스의 Package, Pin count, Speed grade에 따라서 디바이스를 검색해 주는 작업을 하는 부분입니다. 따라서 타겟 디바이스를 쉽게 찾아 선택할 수 있게 도와주는 부분입니다. 아래의 [그림 4-8]에서는 현재 COMBO II에서 사용하는 EP2C35F672C8을 선택한 모습을 보여주고 있습니다.



[그림 4-8] Family & Device Settings

Quartus II는 기본적으로 설계, 합성 및 시뮬레이션 작업을 할 수 있도록 기능을 포함하고 있습니다. 하지만 Quartus II가 PLD 지원을 목적으로 만들어진 설계 프로그램이기 때문에 이러한 개개의 작업에서의 최대의 성능을 지원해 준다고는 말하기 힘듭니다.

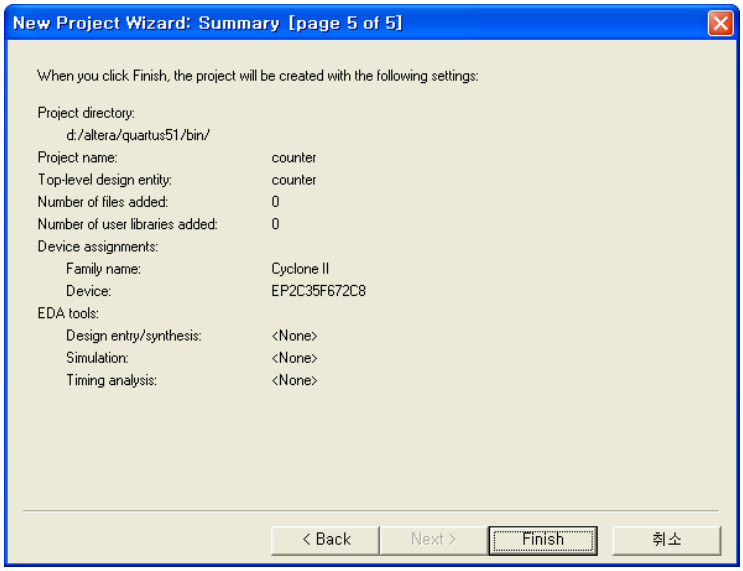
따라서 여기에서는 이러한 점을 보완하기 위해 다른 전문적인 툴과 연동해서 작업이 가능합니다. 이러한 작업을 위해 [그림 4-9]의 창에서 이러한 툴의 연동을 시킬 수 있습니다. 따라서 현재 사용자의 컴퓨터에서 연동하려는 다른 프로그램이 설치되어 있다면 여기에서 설정을 해 주면 됩니다.



[그림 4-9] EDA Tool Settings

여기에서는 현재까지의 프로젝트 설정 단계에서 각 단계에 대한 설정이 어떻게 되었는지 모두 보여주게 됩니다. 따라서 사용자가 이러한 정보를 보고 프로젝트 선언 단계에서 다르게 설정된 부분이 없는지 다시 한번 확인하는 단계가 되겠습니다.

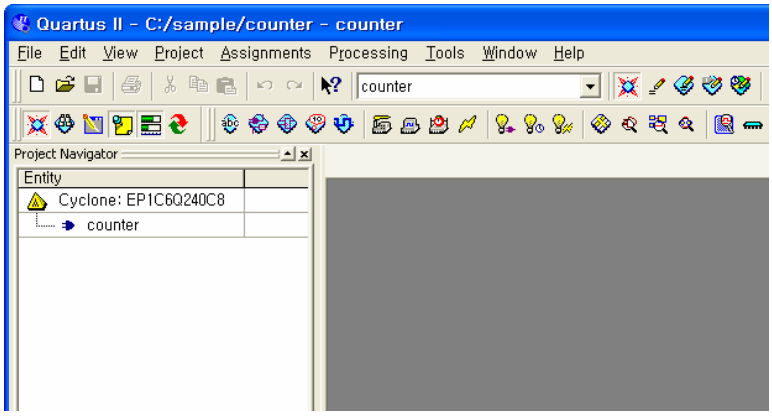
따라서 이러한 확인 절차에 의해 잘못된 부분은 Back 버튼에 의해 그 단계로 가서 수정을 하고, 이상이 없을 시 Finish 버튼에 의해 프로젝트 설정 단계를 마치면 됩니다. 이러한 설정 부분은 프로젝트 설정을 마치더라도 Setting 메뉴를 통해 재 수정이 가능합니다. 아래 [그림 4-10]에서는 Summary 창을 보여주고 있습니다.



[그림 4-10] Summary

이전까지는 작업으로 프로젝트가 선언되었습니다. 이로써 설계할 수 있는 환경이 만들어진 것입니다.

이렇게 프로젝트를 선언하면 [그림 4-11]과 같이 Quartus II 프로그램의 왼쪽 상단의 타이틀 바에 현재 프로젝트가 선언된 경로명 및 프로젝트의 이름이 표시됩니다.

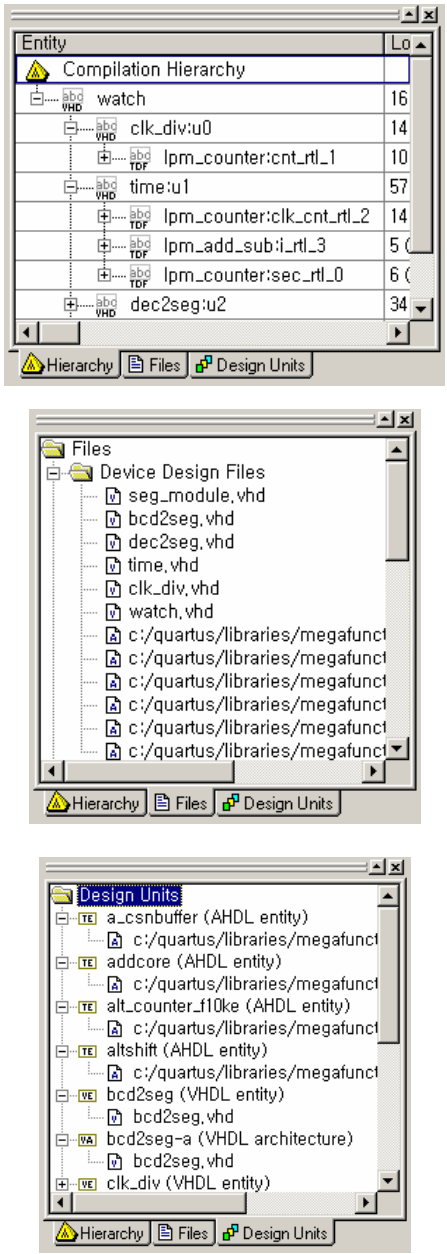


[그림 4-11] 타이틀에 표시된 프로젝트 이름

작업을 진행하는 중에 Project Navigator라고 해서 프로젝트 내에 어떤 형태로 구성 되어 있는지 알아볼 수 있는데 이것은 크게 3가지로 구분됩니다. 프로젝트의 계층구조를 보여주고, Top-Level Design File이라고 해서 최종 Design을 확인할 수 있는 Hierarchy라는 것이 있습니다. 또한 포함되어있는 파일들(시뮬레이션 파일, 프로그래밍 파일, 하위 블

록 및 각 블록에 포함된 파일)을 쉽게 확인하고 관리할 수 있는 것이 Files입니다. 마지막으로 Design Unit 의 type 세부사항을 Design Units에서 확인해볼 수 있습니다.

[그림 4-12]에서 맨 위에 있는 것이 Hierarchy 이고, 중앙에 Files, 아래에 Design Units을 보여주고 있습니다. 이것을 통해 프로젝트의 계층구조를 파악해볼 수 있습니다.



[그림 4-12] Project Navigator

3) Design Entry

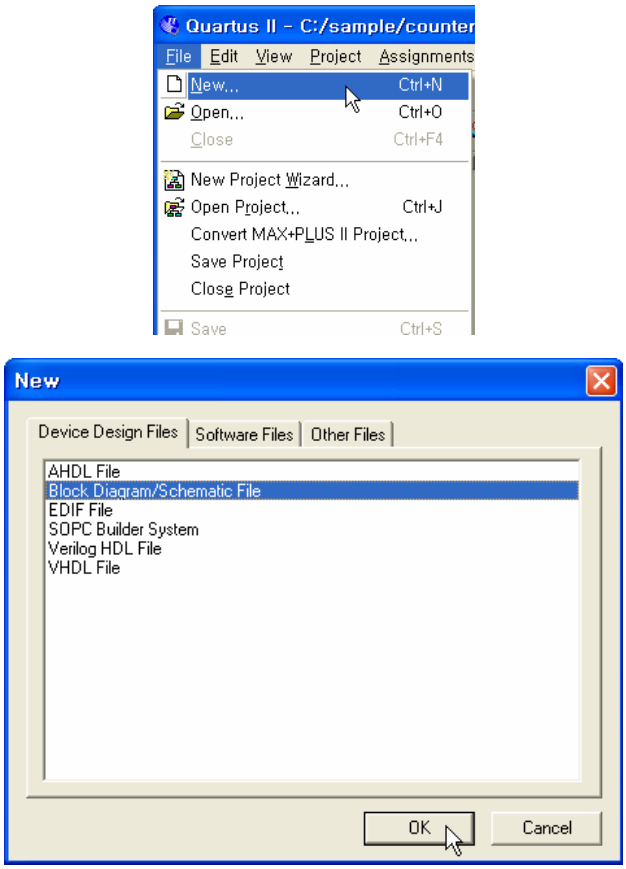
❑ Graphic Editor (Schematic Capture에 의한 설계)

① 특징

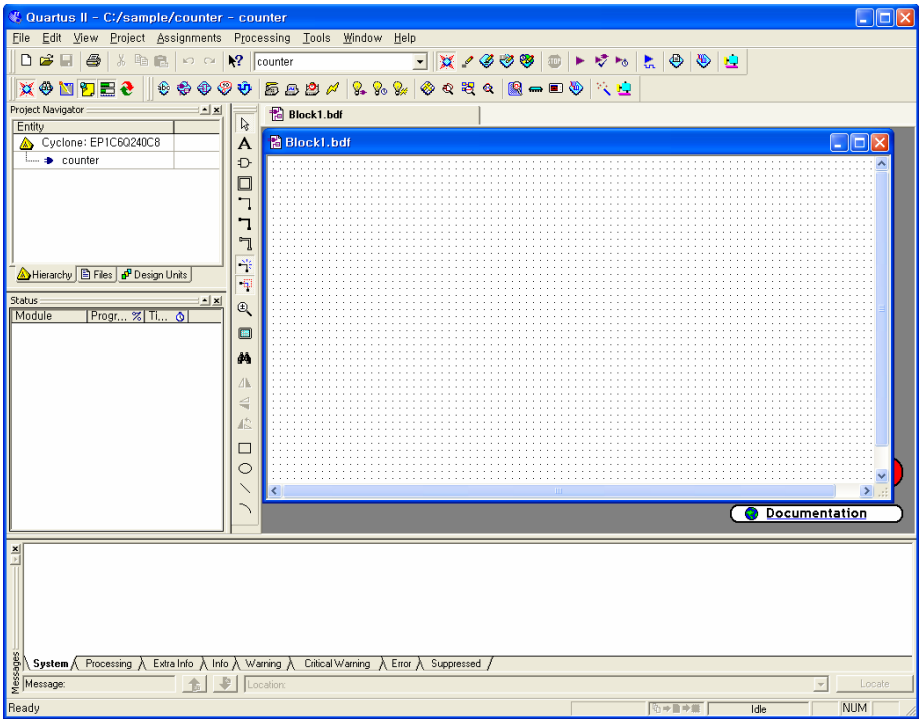
특정 기능을 가진 심볼 라이브러리를 생성하여, 그 심볼 라이브러리의 입 출력 데이터 라인을 선 (wire) 등으로 각 심볼 등의 입 출력 데이터 신호를 연결하여 논리 회로를 설계하는 방법입니다.

② Graphic Editor 선택하기

Graphic Editor를 이용해 하나의 논리 회로를 설계하기 위해서는 Graphic Editor 창을 활성화 시켜야 하는데, [그림 4-13]과 같이 File -> New 메뉴를 선택하여 New 창을 활성화한 후 Block Diagram/Schematic File이라는 항목을 선택하여 OK 버튼을 누르면 [그림 4-14]와 같이 Graphic Editor 창이 활성화 됩니다.



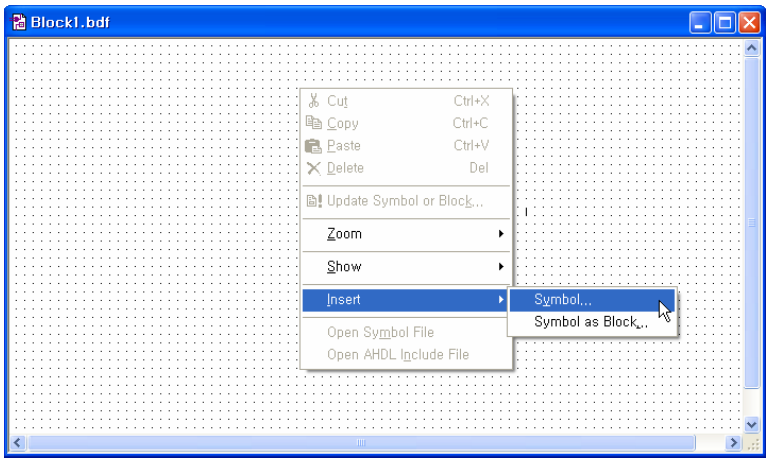
[그림 4-13] 설계 메뉴에서 Block Diagram/Schematic File 선택



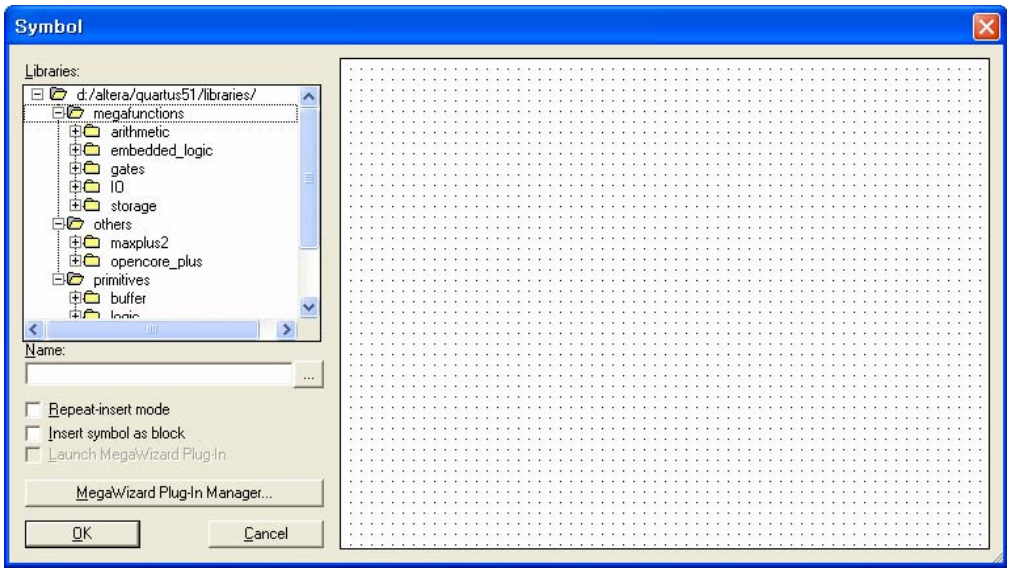
[그림 4-14] Block Diagram/Schematic File을 활성화 한 상태

③ 심볼 라이브러리

Quartus II의 Block Diagram/Schematic File에서는 도면을 마우스 왼쪽 버튼으로 더블-클릭하는 방법 또는 마우스의 오른쪽 버튼을 눌러 Pop-Up 메뉴의 Insert -> Symbol 항목을 선택하는 방법과 Quartus II 메뉴의 “Edit -> Insert Symbol”을 통해 라이브러리를 불러오는 방법이 있습니다.



[그림 4-15] Pop-up 메뉴에서 "Insert Symbol" 선택

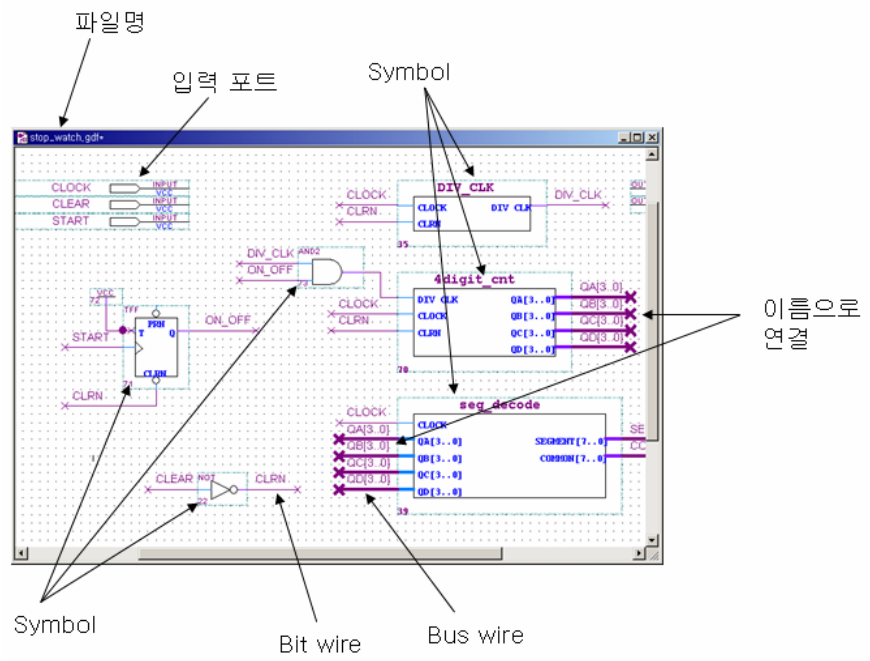


[그림 4-16] Insert Symbol 창

위의 Enter Symbol 창에서 Name에 직접 심볼 명을 쓰거나, 왼쪽의 Libraries에서 해당 심볼 라이브러리가 들어 있는 폴더에서 찾는 방법으로 Symbol들을 검색 할 수 있습니다. 여기에서는 AND 또는 OR 게이트의 간단한 Symbol에서 약간의 논리가 포함된 74 시리즈, Megafunctions라고 하여 기본 설정과 Parameter 값만 변화시켜 줌으로써 쉽게 설계할 수 있도록 해주는 라이브러리입니다. 이처럼 일부 항목을 바꾸어서 범위가 큰 논리 회로를 쉽게 설계할 수 있도록 해줍니다.

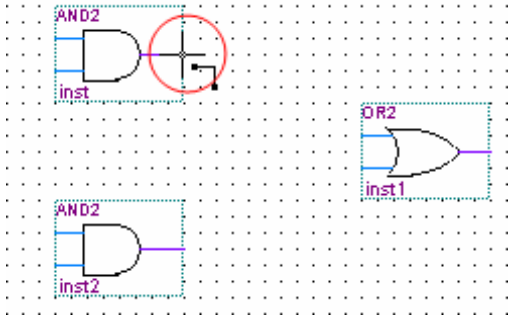
④ 각 Symbol의 연결

각 Symbol 간의 데이터 연결은 wire 또는 이름으로 연결해 줄 수 있습니다. [그림 4-17]은 stop-watch 예제를 보여주고 있다. 각 Symbol 라이브러리 간의 bus 데이터를 연결할 때는 선 굵기가 굵은 bus wire를 사용하고, bit 데이터를 연결할 때는 굵기가 가는 bit wire를 사용합니다.

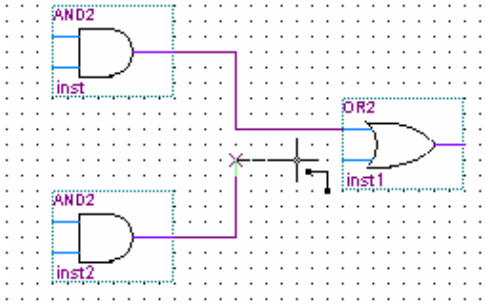


[그림 4-17] Stop-watch 예제에서의 연결

Symbol을 연결하는 방법은 [그림 4-18]과 같이 각 Symbol의 데이터 라인의 연결점을 마우스로 가져가면 마우스의 포인터가 십자 모양으로 바뀝니다. 이 상태에서 마우스의 왼쪽 버튼을 누른 채 드래그 하면 wire가 생성이 되는 데, [그림 4-19]와 같이 wire를 연결하고자 하는 곳까지 드래그 하여 연결하면 됩니다.

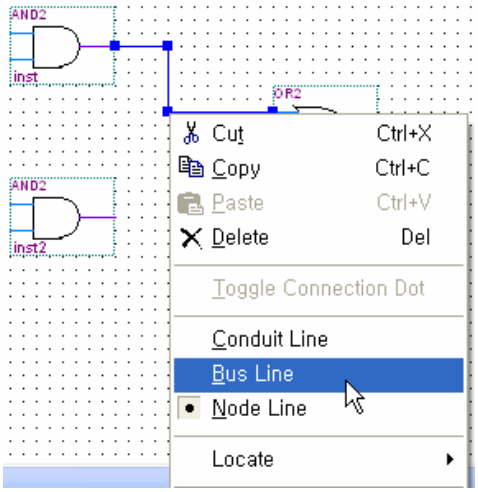


[그림 4-18] 마우스 포인터의 변화



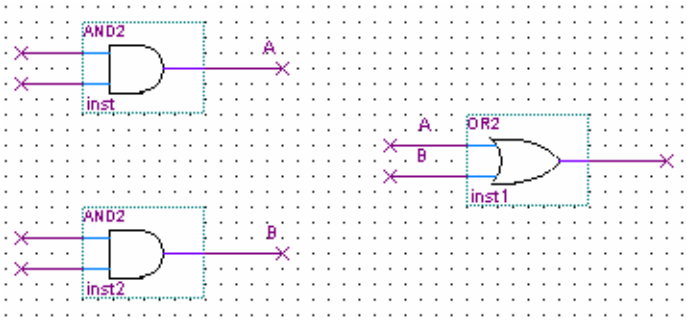
[그림 4-19] wire 연결

bus wire의 경우에는 [그림 4-20] 위에서처럼 wire을 연결한 후, 마우스 왼쪽 버튼의 더블 클릭으로 그 bus wire 전체를 선택 할 수 있습니다. 이 때, 마우스의 오른 쪽 버튼을 눌러 pop-up 메뉴의 Bus Line을 선택하면 됩니다.



[그림 4-20] bus wire로 변환

이름으로 연결할 경우에는 연결하고자 하는 Node를 [그림 4-21]과 같이 Symbol에서 wire을 일정 부분 연결한 후 마우스의 왼쪽 버튼을 이용하여 wire를 선택(클릭)하고 Node 명을 적어 주면 됩니다. 버스 데이터의 경우도 bus wire을 그린 후 이름을 버스의 형태로 적어 줍니다. 이름을 적을 때 4bit의 A라는 이름이 있다면 A[3..0] 또는 A[0..3]로 적어 줍니다. 이 차이는 MSB(상위 비트)의 번호가 4인지 또는 0인지의 차이입니다.

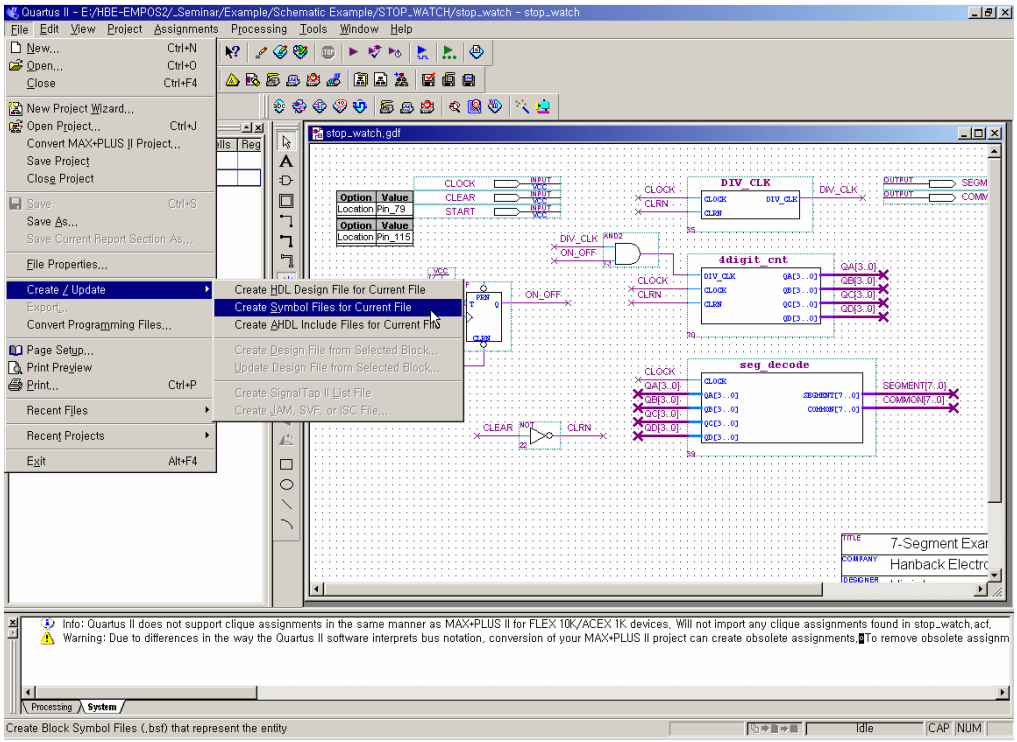


[그림 4-21] 이름으로 연결

⑤ 심볼 생성

사용자가 설계하여 만든 논리 회로를 하나의 블록으로 하여 더 큰 프로젝트에 넣을 수 있습니다. 이렇게 생성한 것은 Graphic Editor 상태에서는 Symbol 형태로 불러오게 됩니다. 이 Symbol 형태를 만들기 위해서는 Graphic Editor를 활성화 시킨 상태에서 File -> Create / Update -> Create Symbol Files for Current File 메뉴를 선택하여 변화시킵니다. 이

진행 과정은 왼쪽의 상태표시 창에서 확인해볼 수 있습니다.



[그림 4-22] 심볼 생성

❑ Text Editor (HDL에 의한 설계)

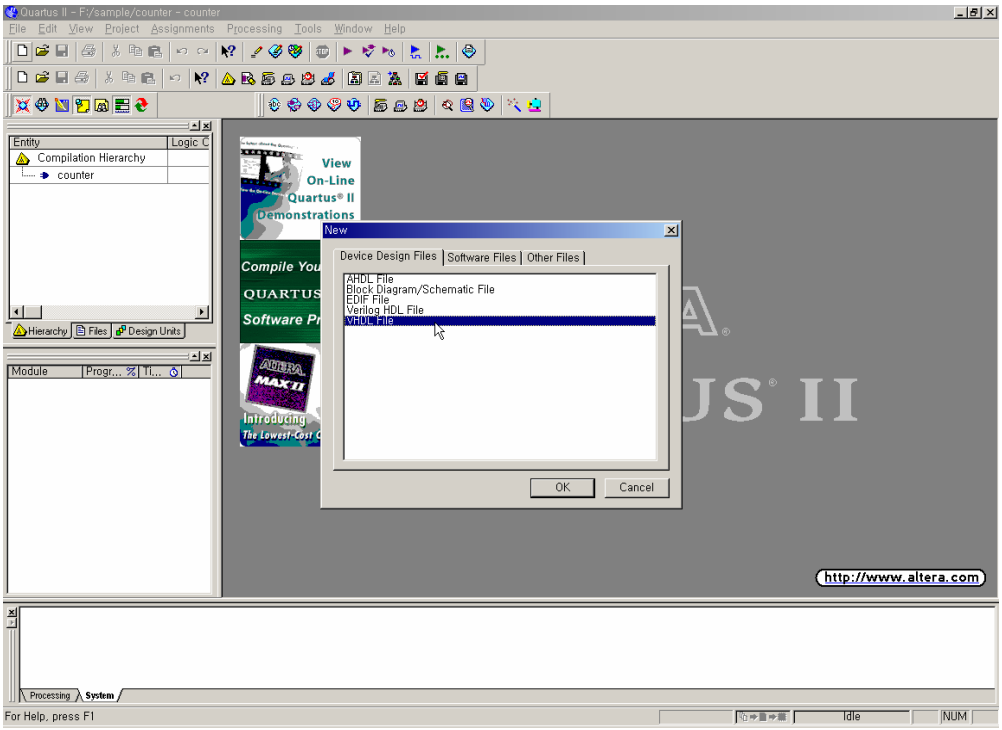
HDL을 이용하여 논리 회로를 설계하는 방법은 각 HDL에 따라 문법을 알아야 하기 때문에, 많은 내용을 설명해야 합니다. 따라서 이 사용자 설명서에서는 가장 기본적으로 Editor 창을 활성화 시키는 방법을 간단한 예제를 보여드리겠습니다.

① 특징

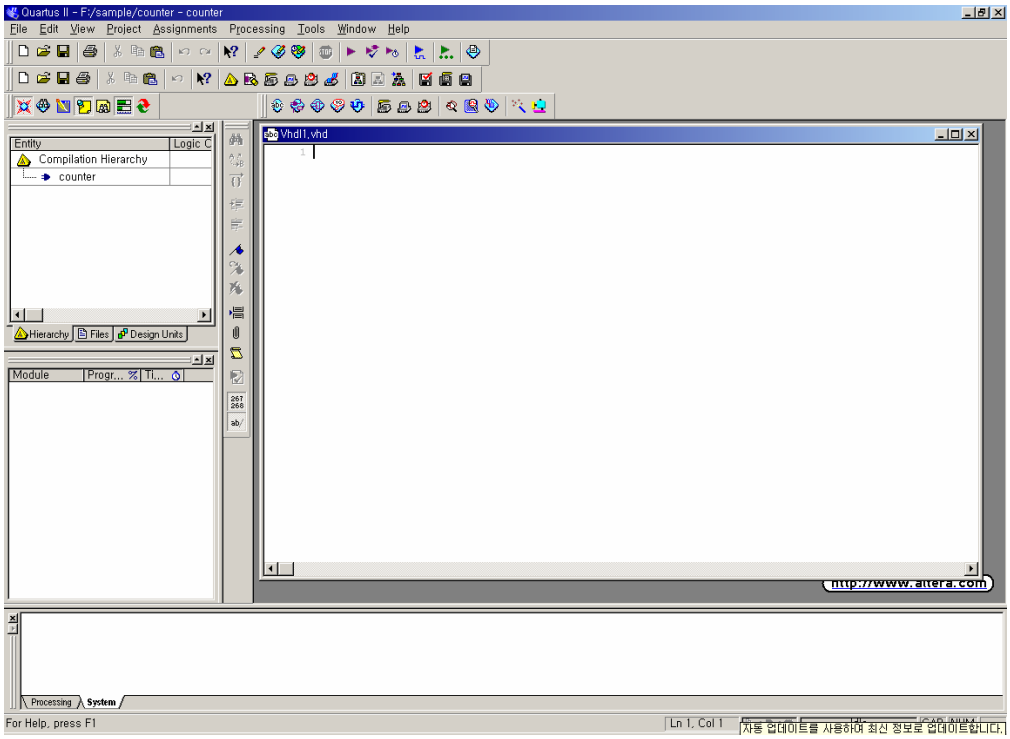
언어를 이용해서 논리 회로를 설계하는 것으로 보통 HDL(Hardware Description Language) 언어를 사용하여 설계합니다. HDL의 종류는 VHDL(Very High-speed integrated circuit HDL), Verilog HDL등이 있는데, Quartus II에서는 Text Editor를 이용하여 HDL을 설계합니다.

② Text Editor 선택하기

Text Editor를 이용하여 하나의 논리 회로를 설계하기 위해서는 Text Editor 창을 활성화 시켜야 하는데, File -> New 메뉴를 선택한 후 [그림 4-24]와 같은 그림이 활성화 되며, 여기에서는 VHDL File을 선택한 것을 보여주고 있다. 이렇게 HDL의 종류를 선택했으면 OK 버튼을 눌러서 Text Editor 창을 활성화 시킨다.



[그림 4-23] New 메뉴에서 Text Editor File 선택



[그림 4-24] Text Editor File을 활성화 한 상태

③ 설계하기

HDL 코드를 사용하여 논리 회로를 설계하기 위해서는 각 HDL (AHDL, VHDL, Verilog HDL 등)의 문법 등에 대해 알아야 할 것이 많기 때문에 간단히 설계한 예만 보여드리겠습니다. 참고로 각각의 설계파일은 다른 확장자(VHDL 파일은 .vhd, AHDL 파일은 .tdf, Verilog HDL 파일은 .v)를 갖는다는 것을 유념해야 합니다.

C:/qdesigns/chiptrip/speed_ch.tdf

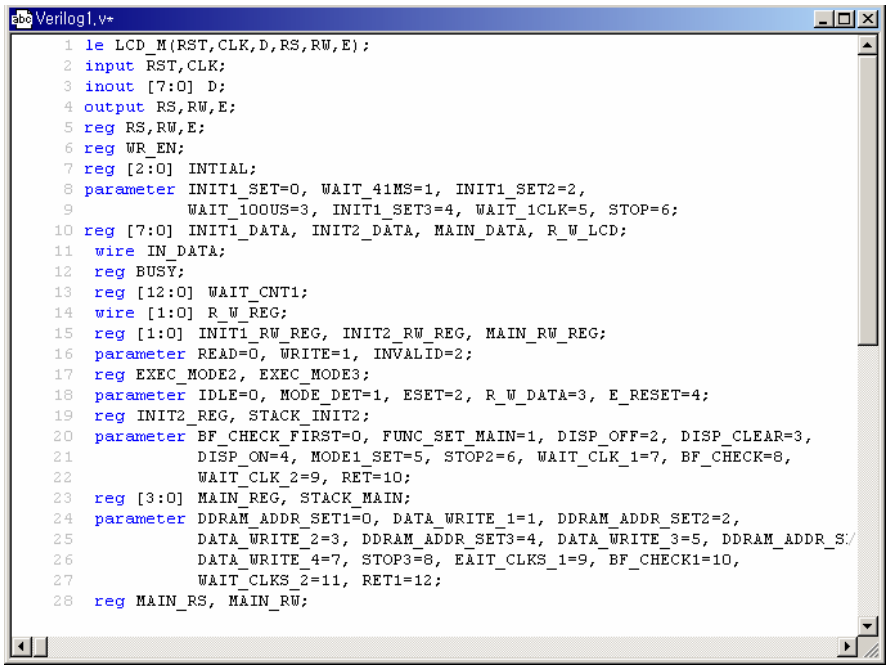
```
SUBDESIGN speed_ch
(
    Accel_in, reset, clk    : INPUT;    % File inputs   %
    get_ticket              : OUTPUT    % File output   %
    % File outputs default to GND unless otherwise specified    %
)

VARIABLE
    speed   :   MACHINE             % Create state machine called   %
                OF BITS (q1,q0)     % speed with bits q1 & q0       %
                WITH STATES (

    legal,          % You are driving at a legal speed      %
    warning,        % Police are giving you a warning!      %
    ticket          % Police are giving you a ticket!       %

    );

    gt  :   dff;    % Define flip-flop called gt %
    spd_out :   node;   % Define node called spd_out %

BEGIN
    speed.clk   = clk;      % Connect clock input of speed to INPUT clk     %
    speed.reset = reset;    % Connect reset input of speed to INPUT reset   %
    gt.clk      = clk;      % Connect clock input of gt to INPUT clk        %
    gt.d        = spd_out;  % Connect d input of gt to node spd_out         %
    get_ticket  = gt.q;     % Connect OUTPUT get_ticket to q output of gt   %

    CASE speed IS

    WHEN legal =>

        IF accel_in THEN
            speed = warning;

        ELSE
```

[그림 4-25] AHDL로 설계한 예

G:/jung1/bin/jung1/counter.vhd

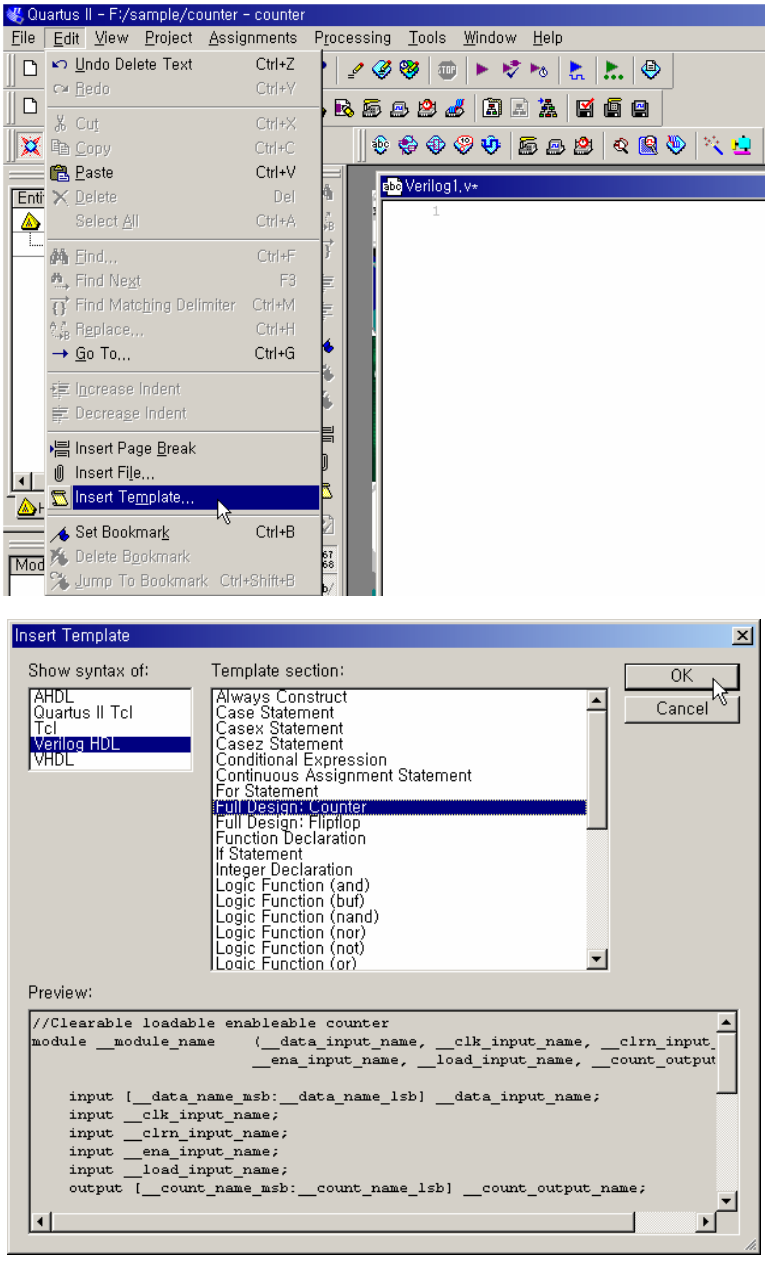
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY COUNTER IS
        PORT(
        CLK     : IN    STD_LOGIC;
        CLEAR   : IN    STD_LOGIC;
        G       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        SEG7    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COUNTER;

ARCHITECTURE ARC OF COUNTER IS
    SIGNAL CNT      : INTEGER RANGE 0 TO 5;
    SIGNAL NUM      : INTEGER RANGE 0 TO 15;
    SIGNAL HOUR     : INTEGER RANGE 0 TO 59;
    SIGNAL MIN, SEC : INTEGER RANGE 0 TO 59;
    SIGNAL H10, H1  : INTEGER RANGE 0 TO 9;
    SIGNAL M10, M1  : INTEGER RANGE 0 TO 9;
    SIGNAL S10, S1  : INTEGER RANGE 0 TO 9;
    SIGNAL CNTS     : INTEGER RANGE 0 TO 499999;
    SIGNAL S_CLK    : STD_LOGIC;
    SIGNAL M_CLK    : STD_LOGIC;
    SIGNAL H_CLK    : STD_LOGIC;

COMPONENT SEP
    PORT(
            A       :IN   INTEGER RANGE 0 TO 63;
            TEN, ONE:OUT  INTEGER RANGE 0 TO 15);
END COMPONENT;

COMPONENT DEC7
    PORT(
            BCD       :IN   INTEGER RANGE 0 TO 15;
            D         :IN   STD_LOGIC;
            SEG       :OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
```

[그림 4-26] VHDL로 설계한 예



[그림 4-27] Verilog HDL로 설계한 예

참고. 설계하는 문법은 Text Editor 창을 활성화 시킨 상태에서 [그림 4-28]과 같이 Template 메뉴의 각 언어 - VHDL, Verilog HDL등 - 의 Template 창을 선택하여 설계하는 구문의 문법을 알 수 있습니다.



[그림 4-28] Template 선택

4) Compile

① 역할

컴파일러는 프로젝트에 포함된 모든 디자인 파일을 통합하는 과정과 문법 오류와 일

반적인 설계의 오류를 찾는 과정을 하게 됩니다. 또한 논리 회로 합성 및 설계한 논리 회로가 하드웨어에서 이루어지는 가장 기본 단위인 로직 셀로 나누고, 각 셀의 위치를 결정하는 과정을 하게 되고, 시뮬레이션과 시간 측정을 할 수 있도록 하는 파일을 생성하고, 마지막으로 타겟 디바이스에 프로그래밍 할 수 있는 파일을 만드는 과정을 진행합니다.

컴파일 과정은 보통 2번의 과정을 수행하게 되는데, 처음의 컴파일 과정은 현재 작성된 설계 파일에 대한 오류를 검증하는 단계입니다. 따라서 이 과정을 통해 컴파일 오류를 찾아서 수정이 가능합니다. 다음의 컴파일 과정은 설계 파일에 대한 핀 할당 정보 등의 프로젝트 선언 다음의 수정 내용을 현재 프로젝트에 적용해 주는 단계라 할 수 있습니다.

② Compiler

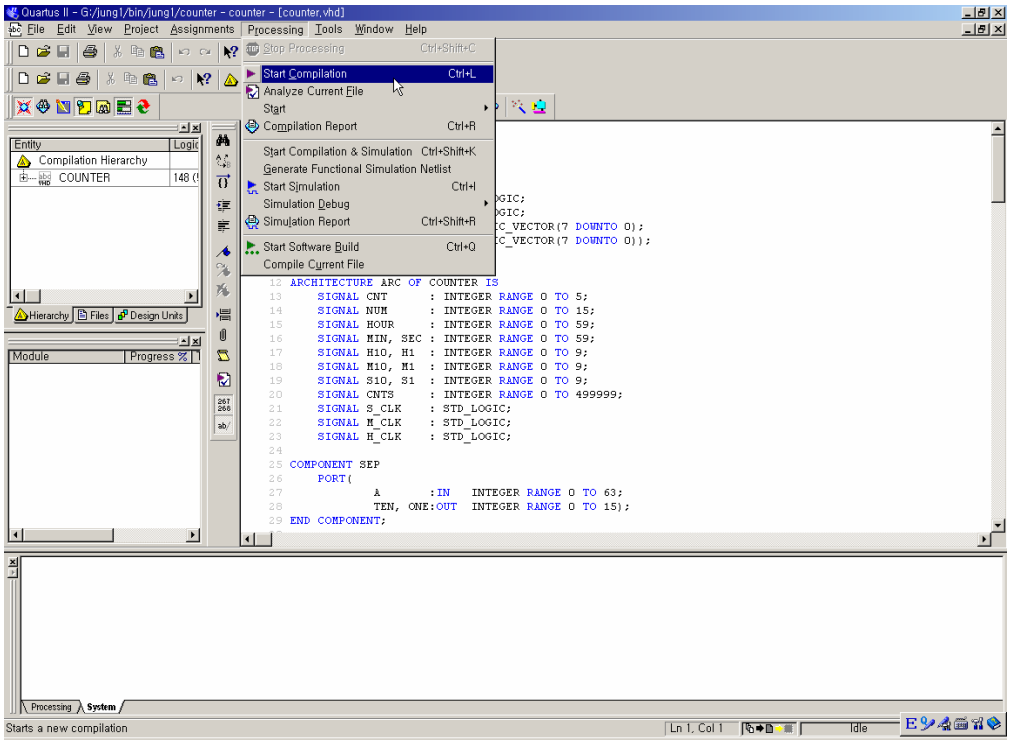
[그림 4-29]와 같이 메뉴의 Processing -> Start Compilation 항목을 선택하면 컴파일 과정을 수행하게 됩니다. 이 컴파일 과정은 4단계로 나뉘어 수행하게 되는데, 왼쪽의 상태 표시 창에서 진행 상황을 확인해 볼 수 있습니다.

Analysis & Synthesis 에서는 프로젝트 내에 있는 파일들을 분석하고 통합하는 과정에서 발생하는 기본적인 오류를 체크하는 과정입니다. 따라서 회로를 분석하는 과정에서 오류가 발생하면 컴파일을 중단하고 메시지 창에 에러를 찾아 표시해 줌으로써 회로를 쉽게 수정할 수 있게 해줍니다.그리고 회로를 분석하는 과정에서 디바이스의 내부구조를 효율적으로 사용하기 위해 불필요한 논리를 제거합니다.

Fitter 에서는 사용자가 디바이스를 정해주지 않았을 때는 Quartus II에서 회로에 최적인 디바이스를 자동적으로 찾아서 디바이스의 로직 셀에 배치해 줍니다. 디바이스를 정해 주었을 때는 그 디바이스에 맞게 회로를 적당한 로직 셀에 배치합니다.먼저 각각의 논리 함수를 가장 적합한 위치의 로직 셀에 할당하고 상호 연결이나 핀 등을 적합하게 할당시킵니다.

Assembler에서는 이전의 작업을 디바이스에 프로그램 할 수 있는 파일을 생성시킵니다.

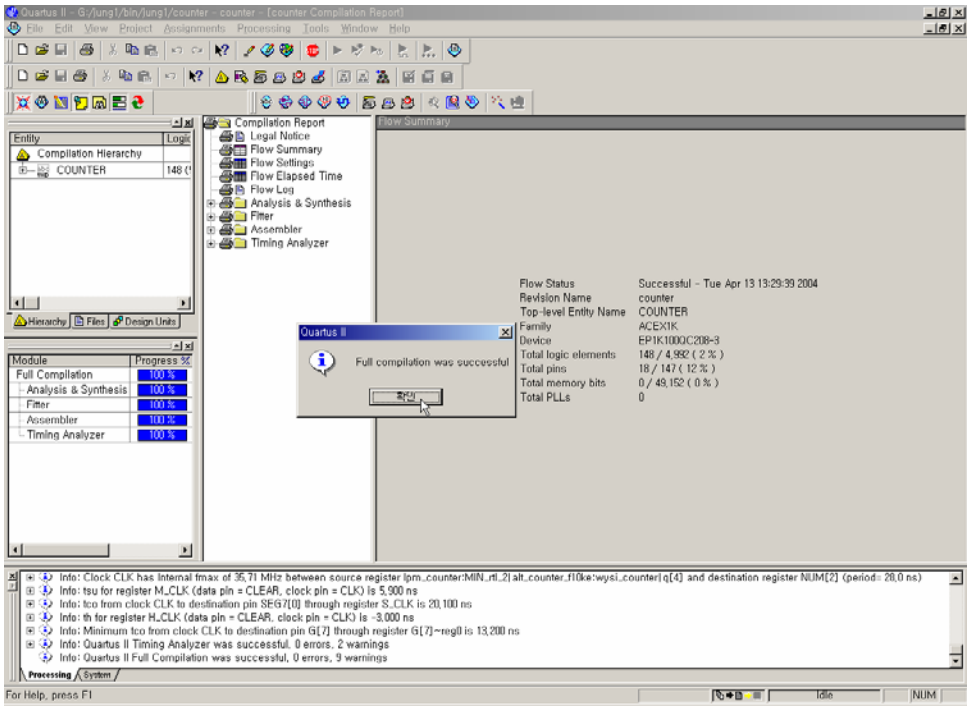
마지막으로 Timing Analyzer에서는 최종적으로 디바이스에 할당된 정보를 가지고 핀과 핀 또는 로직 셀 사이의 데이터 전달 시간을 검사하는 과정입니다.



[그림 4-29] Processing -〉 Start Compilation 항목 선택

[그림 4-30]에서는 컴파일 과정이 끝난 화면을 보여주고 있습니다. 따라서 왼쪽의 상태 표시 창에서 모든 과정이 100%로 완료된 것을 볼 수 있고, 이전에서 말한 4가지의 컴파일 과정을 오른쪽 메인 화면 창에서 리포트 형태로 볼 수 있습니다. 따라서 여기에서 어떠한 형태로 작업이 수행되었는지 나중에 확인해 볼 수 있습니다.

또한 밑에 메시지 창에서 컴파일 과정에 생긴 간단한 정보나, 에러, 또는 경고 등을 표시해 줍니다. 에러 메시지의 발생 시 컴파일이 중단되고 어떠한 내용의 에러 메시지인지 표시해 주게 됩니다. 따라서 에러 메시지를 클릭하면 에러가 난 지점으로 커서가 이동하게 되고 사용자는 에러 메시지와 더불어 에러에 대한 수정을 쉽게 할 수 있습니다.



[그림 4-30] Compiler 과정 완료

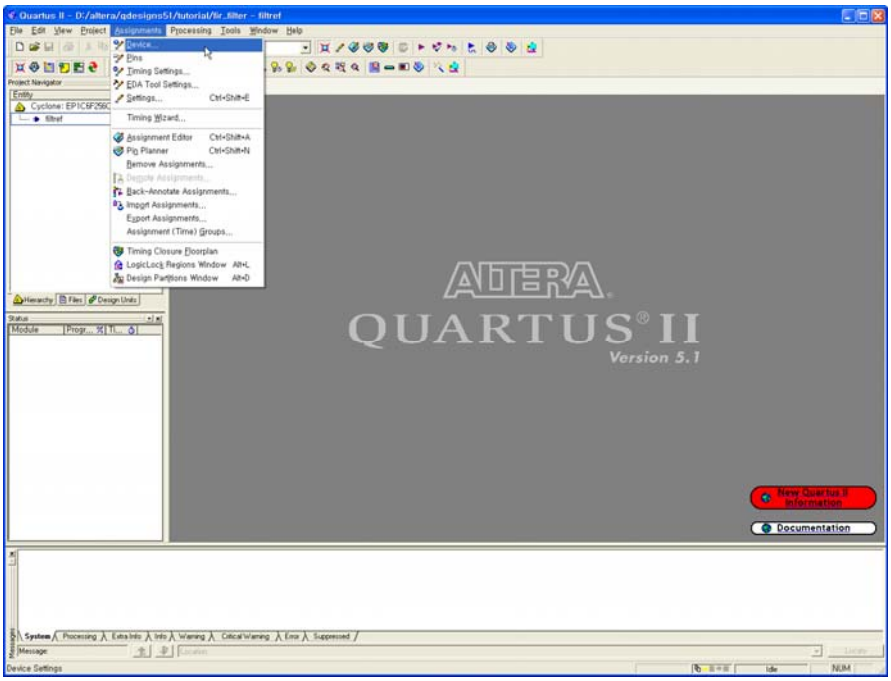
5) Assignments

① 역할

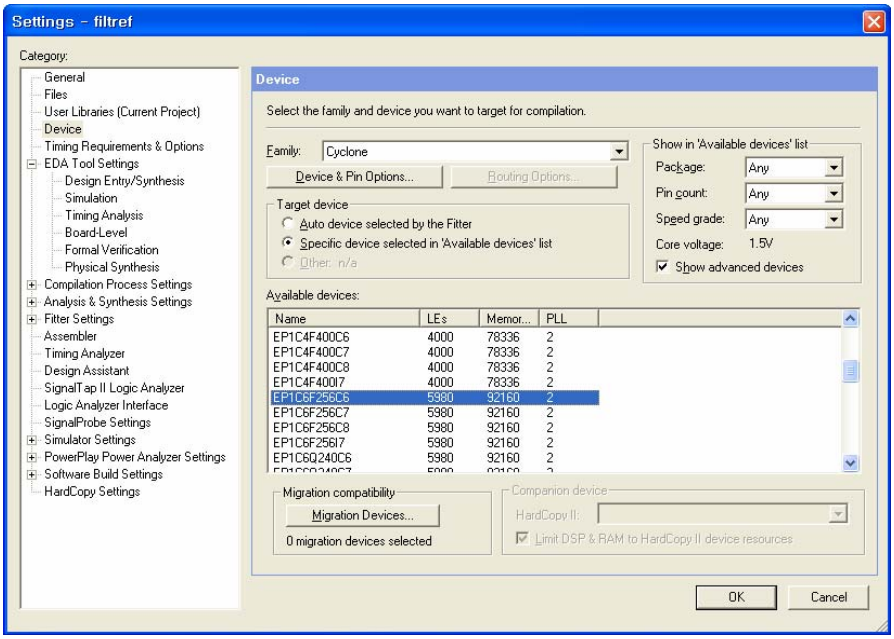
이 작업은 현재 작업하는 프로젝트의 디바이스 설정과 선택한 디바이스의 I/O 핀 할당에 대한 작업을 수행하게 됩니다. 디바이스 선택은 사전에 프로젝트 선언 단계에서 현재 사용하는 디바이스를 선택했습니다. 하지만 프로젝트 선언 후에 이 메뉴를 통한 디바이스 수정 작업을 해 줄 수 있습니다. 지금은 Assignments라는 작업에서의 크게 할 작업은 설계 파일과 디바이스에 있는 I/O핀과의 연결을 해 주는 작업입니다. 따라서 사용자가 어떠한 핀을 이용하여 사용할지를 프로젝트에 선언해 주는 작업이 되겠습니다.

②Device

디바이스는 [그림 4-31]과 같이 Assignments -> Device 메뉴를 선택하여 설정합니다. 이 메뉴를 선택하면 [그림 4-32]과 같은 Settings 창이 활성화 됩니다. 이 부분에서 사용하려는 디바이스의 선택을 하거나 다른 디바이스를 찾아 재 설정을 해 줄 수 있습니다.



[그림 4-31] Assignments -〉 Device 항목 선택



[그림 4-32] Device Settings

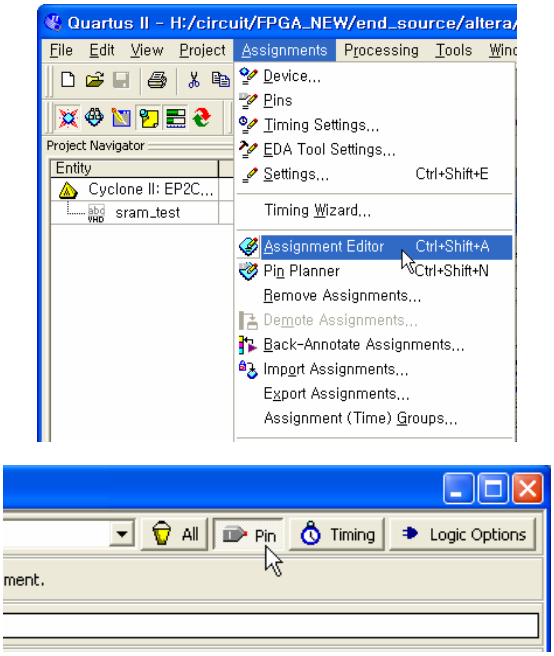
작업 창은 프로젝트 설정에서의 디바이스 설정 과정과 비슷한 구조로 되어 있으며, 동

일한 방법을 통해 디바이스를 선택해 줍니다. 여기에 있는 Device & Pin Options은 디바이스 내부에 있는 특수 핀, 사용하지 않는 핀, 전원 관리 등의 설정을 해 주는 작업을 하게 됩니다. 따라서 디바이스 관리에 대한 전반적인 작업을 설정해 줄 수 있는 메뉴가 되겠습니다.

활성화 된 창에서는 디바이스 선택 이외의 현재 프로젝트에서 사용자가 설정하려는 모든 작업을 할 수 있습니다. 따라서 프로젝트에서 관리하는 파일, 연동해서 사용하는 툴 설정, 컴파일 및 시뮬레이션 과정의 설정 등의 작업을 Settings 창에서 할 수 있습니다.

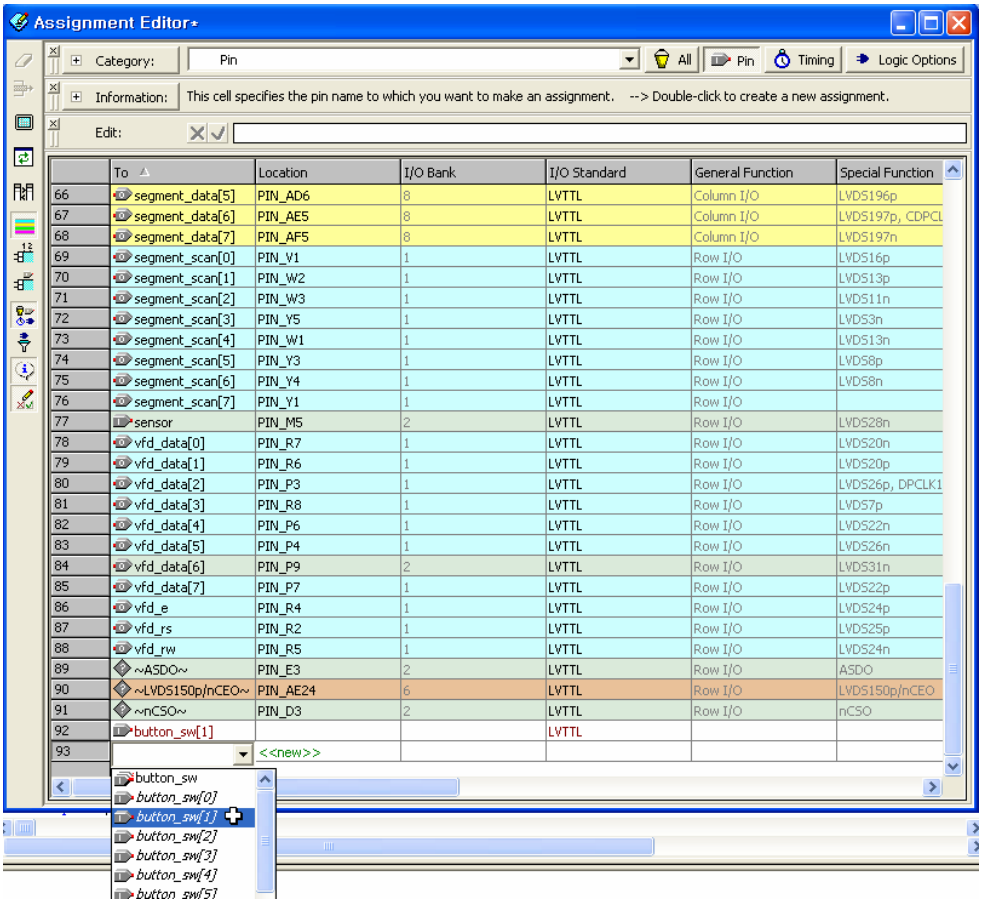
③ Pin Assignments

핀 설정은 [그림 4-33]에서와 같이 메뉴에서 Assignments -> Assignment Editor에서 핀 설정을 해 줄 수 있습니다.



[그림 4-33] Assignment Editor 선택

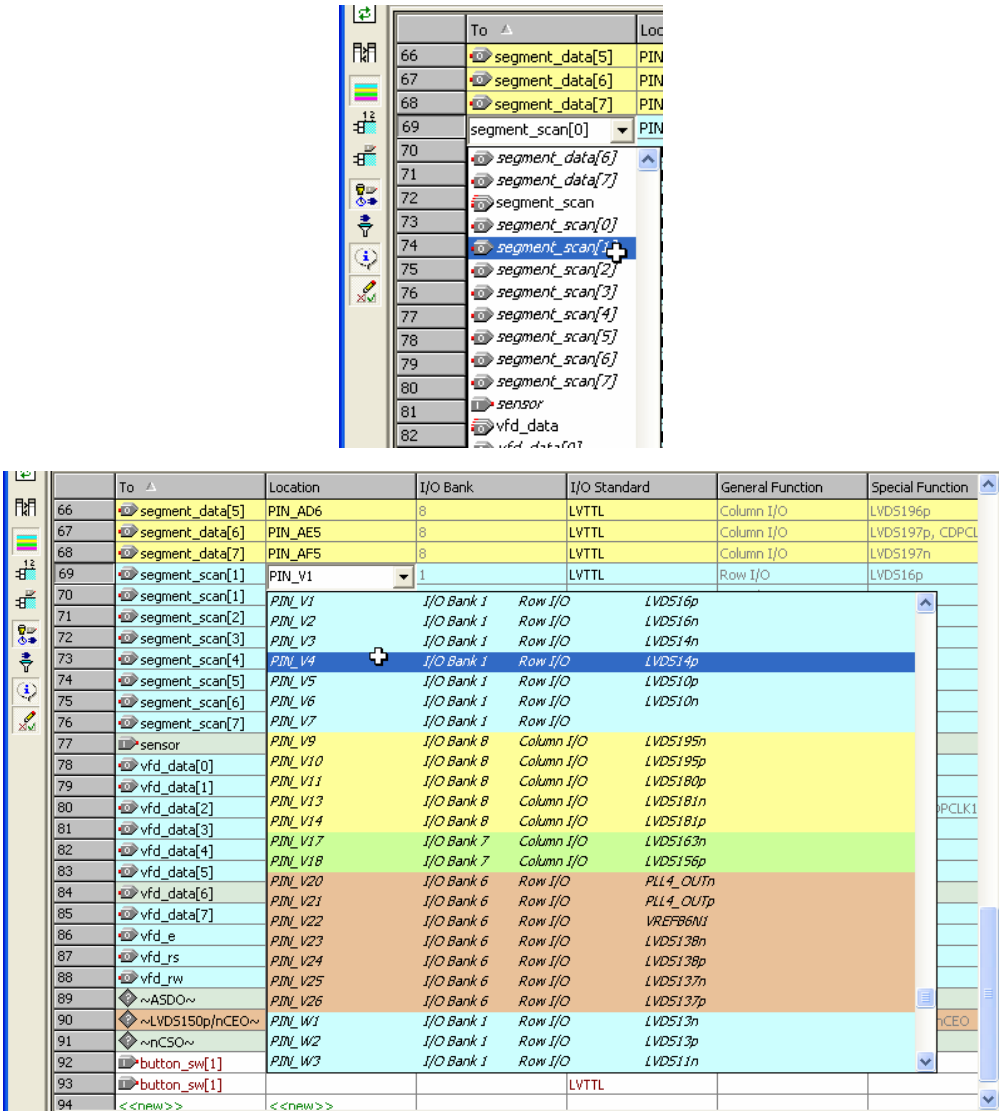
위의 그림에서는 위쪽 그림은 Assignment Editor에 들어가는 메뉴를 보여주고 있습니다. 따라서 이렇게 메뉴에 들어가서 핀 할당을 위한 작업으로 오른 쪽 그림과 같이 Pin 이라는 버튼을 눌러 핀 할당을 위한 환경을 만들어 주어야 합니다.



[그림 4-34] Assignment Editor

위의 [그림 4-34]에서는 Assignment Editor창을 보여주고 있습니다. 여기에서는 할당하려는 핀 이름은 설계 파일 작성 시 미리 지정을 해 놓았습니다. 따라서 핀 이름에 대해서는 따로 타이핑 할 필요 없이 찾아서 Assign해 주면 됩니다.

핀 할당 방법은 [그림 4-34]와 같이 To에서는 핀 이름이 들어가는 부분이고, Location 부분에서는 디바이스의 핀 번호가 들어가게 됩니다. 따라서 To와 Location 부분 하위에 "<<new>>"라는 녹색의 부분에 마우스로 클릭하면, To에서는 설계 파일의 포트 이름이 보이게 되고 Location에서는 선택된 디바이스의 핀 번호 모두가 보이게 됩니다. 따라서 이러한 방식으로 포트 이름과 디바이스 핀 번호를 찾아서 할당해 주면 됩니다. [그림 4-35]에서는 핀 할당에 대한 작업을 자세히 보여 주고 있습니다.



[그림 4-35] 핀 할당 작업

6) Simulation

① 역할

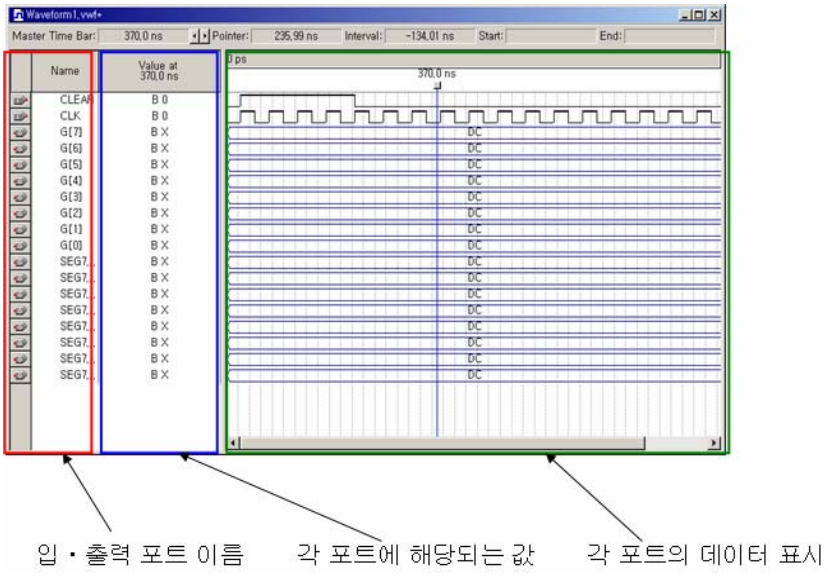
설계한 논리 회로를 하드웨어를 이용해서 검증하는 경우에, 처리하는 데이터량이 많거나 속도가 빠를 경우에는 검증하는데 시간이 많이 걸리고 그 원인을 파악하지 못하는 경우가 있기 때문에 소프트웨어 적으로 먼저 검증하는 것이 필요합니다. Quartus II에서는 이를 위해 파형으로 검증하는 시뮬레이션을 제공하는데, 이 시뮬레이션에서 동작하

는데 필요한 입력 파형의 조건을 설정해 주면 그에 따른 출력 결과의 파형이 출력되어 쉽게 확인해 볼 수 있습니다.

② 구성

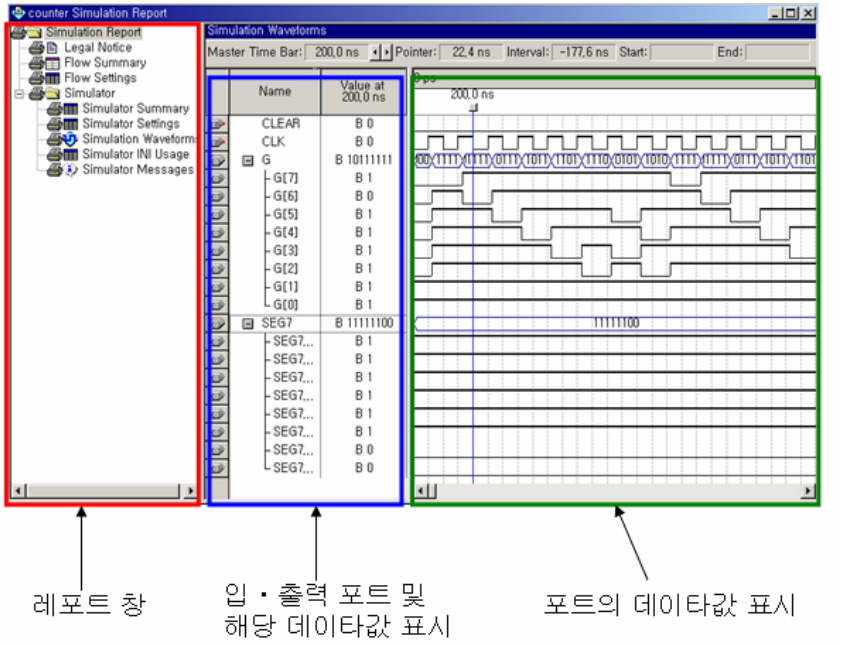
시뮬레이션을 하기 위해서는 입력 파형의 조건을 설정해 주어야 한다고 했습니다. [그림 4-36]에서 이러한 작업을 할 수 있는데, 구성으로 왼쪽에 보이는 창에는 입 출력 포트의 이름을 보여줍니다. 이곳에서는 현재 프로젝트의 설계 파일에서 선언한 포트들이 여기에 나타나는 것입니다. 다음으로 가운데 창에서 각 포트에서 사용자가 선택한 시뮬레이션 시간에 나타나는 출력 값을 표시해줍니다.

따라서 파형을 일일이 보면서 값을 알아내는 것과는 달리, 상태 표시 바를 이동하여 사용자가 측정 하고 싶은 시뮬레이션 시간에 출력되는 포트 값을 이곳에서 표시해줍니다. 마지막으로 오른쪽 창에서 각 포트에 대한 실제 데이터 값을 시간의 따라 파형으로 표시해줍니다. 따라서 이 시뮬레이션 창을 통해 포트들이 어떻게 동작했는지를 여기에서 파형으로 한 눈에 파악할 수 있습니다.



[그림 4-36] Simulation 입력 창

다음으로 [그림 4-37]에서는 시뮬레이션 결과값이 나타나는 창을 보여주고 있습니다. Quartus II 에서는 입력 포트의 입력 값을 설정해 주는 창에서 바로 그 결과값이 나타나지 않고, 새로운 창에서 시뮬레이션 결과 값이 어떻게 나타나는지를 확인할 수 있습니다. 그림과 같이 시뮬레이션 결과 파형이 나오는 창의 구성도 리포트 구간이 더 추가된 것을 볼 수 있습니다.



[그림 4-37] Simulation 결과 창

리포트 창에서는 시뮬레이션 결과에 대한 내용을 Text 형식으로 보여주고 있습니다. 여기에서는 시뮬레이션 과정과 설정 정보 등을 확인해 볼 수 있습니다. 그리고 나머지 두 개의 영역에서 시뮬레이션 결과에 대한 것을 파형으로 검증해 볼 수 있습니다.

③ Simulator

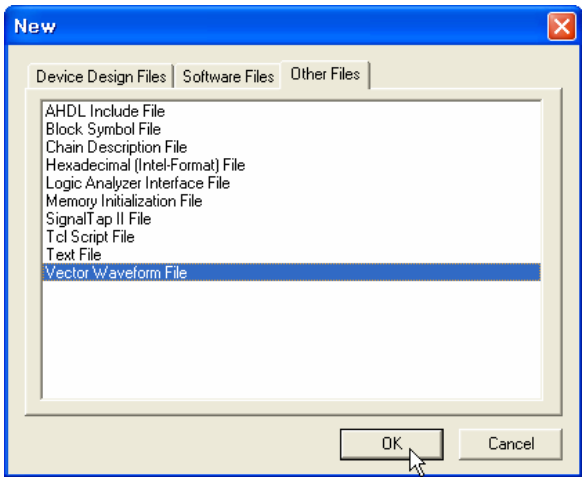
시뮬레이션은 결과 값의 현태에 따라 크게 두 가지 방법으로 나뉘어 집니다. 기능적인 시뮬레이션(Functional Simulation)과 시간 특성을 지닌 시뮬레이션(Timing Simulation)입니다. 기능적인 시뮬레이션은 설계한 회로에 대한 논리적으로 분석하여 이 사항을 시뮬레이션 결과로 보여주게 됩니다. 위의 기능적인 시뮬레이션이 단지 설계한 회로를 논리적으로 검증한다면, 타이밍 시뮬레이션 (시간 특성을 지닌 시뮬레이션 : Timing Simulation)은 설계자가 결정한 하드웨어의 상태 즉, 사용하려고 하는 디바이스와 핀 번호 등의 지연 시간을 고려한 파형으로 검증하는 것입니다.

시뮬레이션은 기본 설정이 타이밍 시뮬레이션으로 동작하게끔 되어 있습니다. 따라서 아무 설정 없이 시뮬레이션을 시키면 타이밍 시뮬레이션으로 동작을 하게 됩니다.

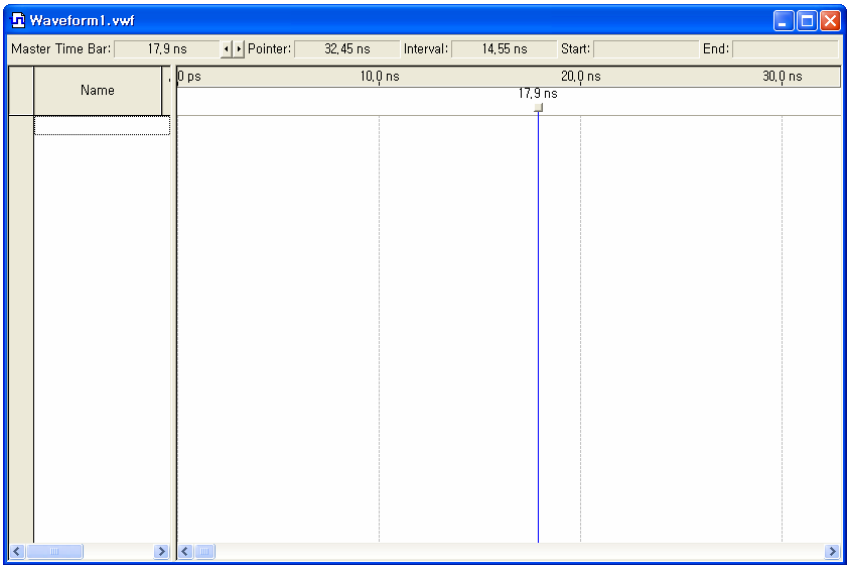
초반에 타이밍 시뮬레이션을 통한 검증에서 타이밍 정보와 논리적인 결과값이 동시에 나옵니다. 또한 포트의 수가 늘어날 때 시뮬레이션에 대한 검증 작업에 어려움이 있을 수 있습니다. 시뮬레이션 분석 방법으로 처음 설계 파일의 검증을 위한 것이므로 기능적인 시뮬레이션을 통해 먼저 설계된 파일의 문제가 없는지 먼저 분석하고 이러한 분

석이 완료가 되었을 때, 타이밍 시뮬레이션을 통해 타겟 디바이스를 적용한 시뮬레이션 검증을 하는 방법으로 이루어 져야 합니다.

다음으로 시뮬레이션 과정을 하나씩 해 보도록 하겠습니다. 먼저 메뉴 창에서 File -> New 항목을 선택하면 [그림 4-38]과 같이 New 창이 활성화 되는데 여기서 Other Files에서 Vector Waveform File을 선택합니다. 그러면 [그림 4-39]와 같이 시뮬레이션을 할 수 있는 Waveform 창이 활성화 됩니다. 여기에서 입력 핀에 파형을 인가해, 출력되는 파형을 검증해 볼 수 있습니다.

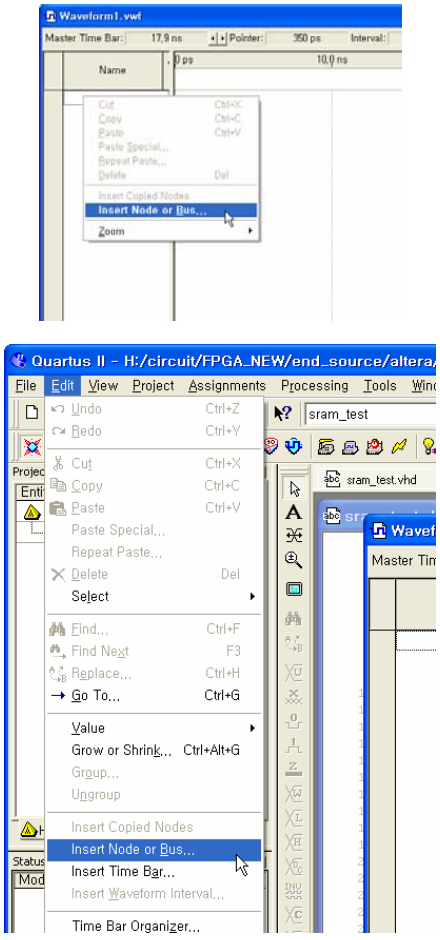


[그림 4-38] Simulation 메뉴 선택

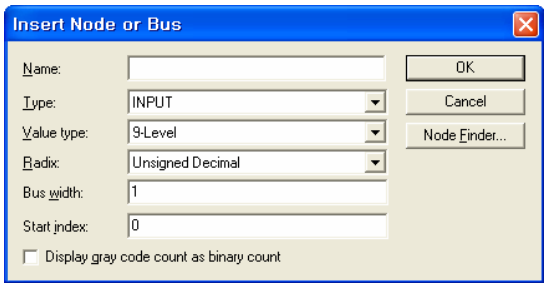


[그림 4-39] Simulation 창 활성화

위의 그림과 같이 시뮬레이션 창을 활성화 시켰으면, 여기에 입 출력 핀을 불러와야 합니다. 불러오는 방법은 [그림 4-40]의 위쪽 그림과 같이 시뮬레이션 창의 왼쪽 편에 있는 Name 창에서 마우스 오른쪽 키에 의한 pop-up메뉴에서 Insert Node or Bus를 선택하거나, [그림 4-40]의 아래 쪽 그림과 같이 Quartus II 메뉴의 Edit -> Insert Node or Bus를 선택합니다. 그러면 [그림 4-41]과 같이 입 출력 포트를 넣는 새로운 창이 생깁니다. 여기서 포트를 검색하기 위해 오른쪽에 있는 Node Finder 버튼을 클릭하여 입 출력 포트를 찾을 수가 있습니다.

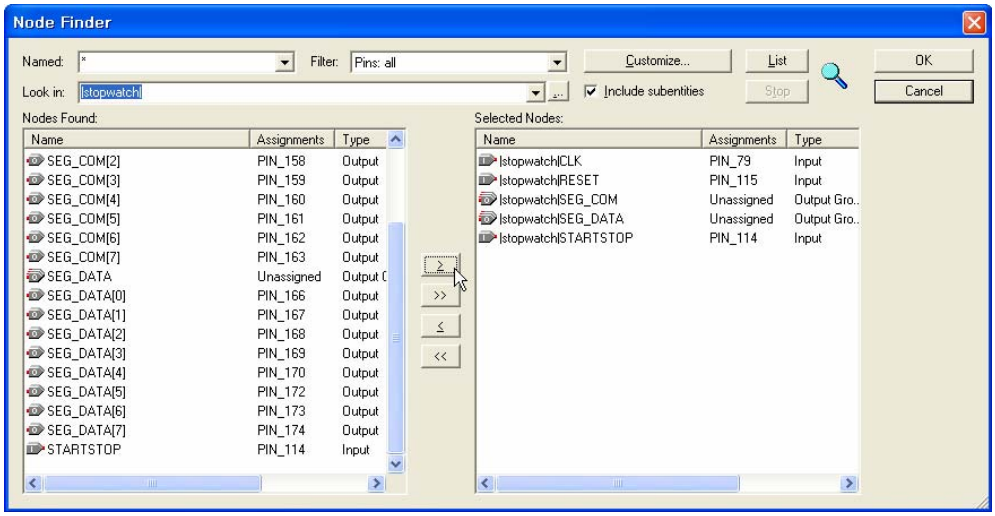


[그림 4-40] Simulation 입 · 출력 포트 불러오기



[그림 4-41] Insert Node or Bus

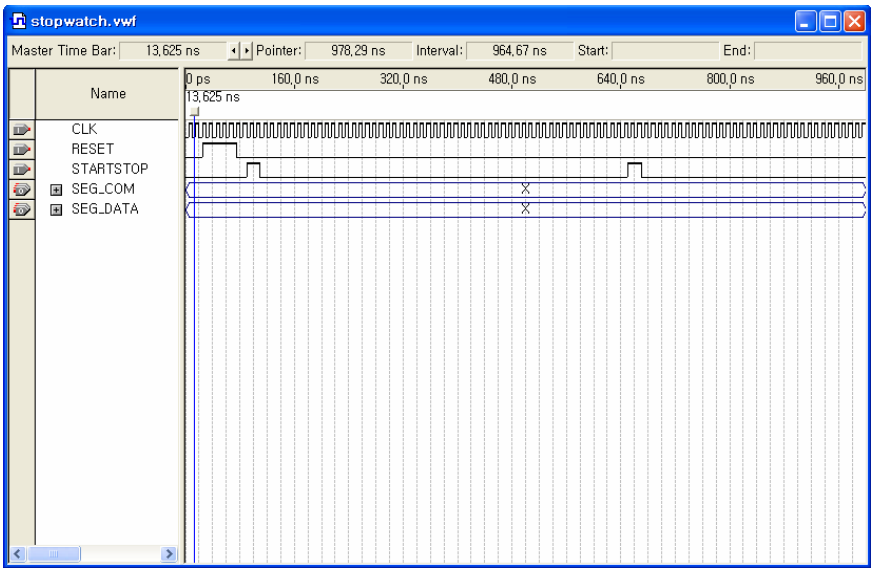
[그림 4-41]의 Node Finder 버튼을 사용하여 [그림 4-42]의 Node Finder 창이 활성화 됩니다. 여기에서 List 버튼을 통해 그림과 같이 설계한 프로젝트의 입 출력 포트가 나타나게 됩니다. 이 포트들 중에 확인하려는 포트를 찾아서 선택을 하면 됩니다.



[그림 4-42] Node Finder

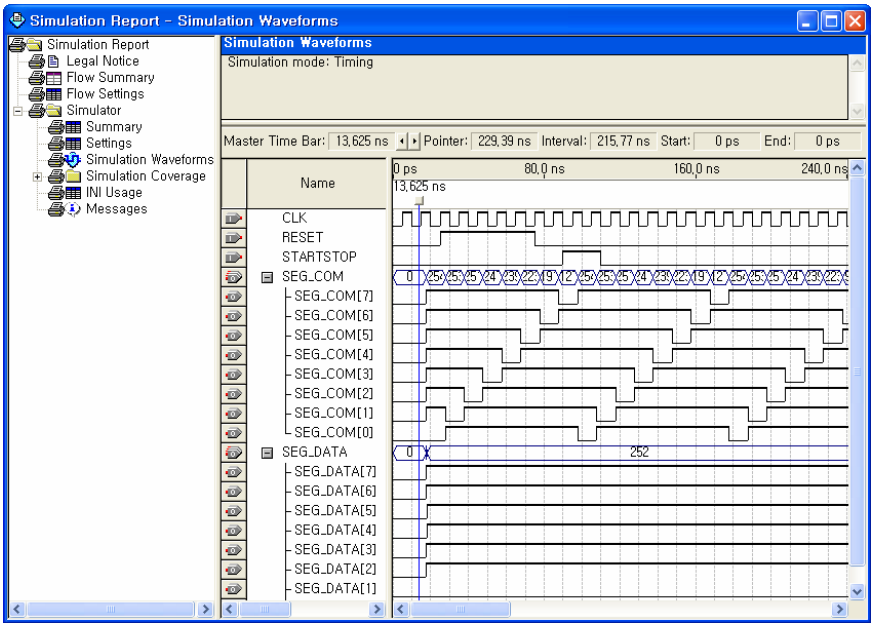
이상으로 포트 선택을 완료하면 OK 버튼을 눌러 마치면, 시뮬레이션 창에 포트 이름이 보입니다. Waveform 창에서 불러들인 포트들을 마우스로 블록화해서 파형을 그려주면 됩니다. 방법으로는 마우스로 파형을 입력하려는 포트의 영역을 블록을 지정하고, File -> Value 에 있는 메뉴를 통해 블록으로 지정된 곳에 입력 파형을 줄 수 있습니다. 여기에서는 0, 1값부터, High Impedance, Unknown, Clock등의 값을 줄 수 있습니다.

그림 [그림 4-43]은 입력 파형에 대한 정의를 마친 모습을 보여주고 있다. Clock은 주기적인 클럭을 준 모습이고 “STARTSTOP”과 “RESET”에는 적당한 구간에서 high 값을 준 모습입니다.



[그림 4-43] Waveform Editor 파형 정의

이렇게 파형 정의를 마치면, Processing -> Start Simulation을 통해 시뮬레이션 결과 파형을 볼 수 있습니다. [그림 4-44]는 이러한 결과 파형을 보여 주고 있습니다. 여기에서는 초기의 파형 입력 창과 달리 별도의 창이 활성화 되어 결과 파형을 분석해 주고 있습니다. 왼쪽의 창의 리포트 창을 통해 시뮬레이션에서 발생하는 결과값을 문서로 표시해 주고 있습니다.



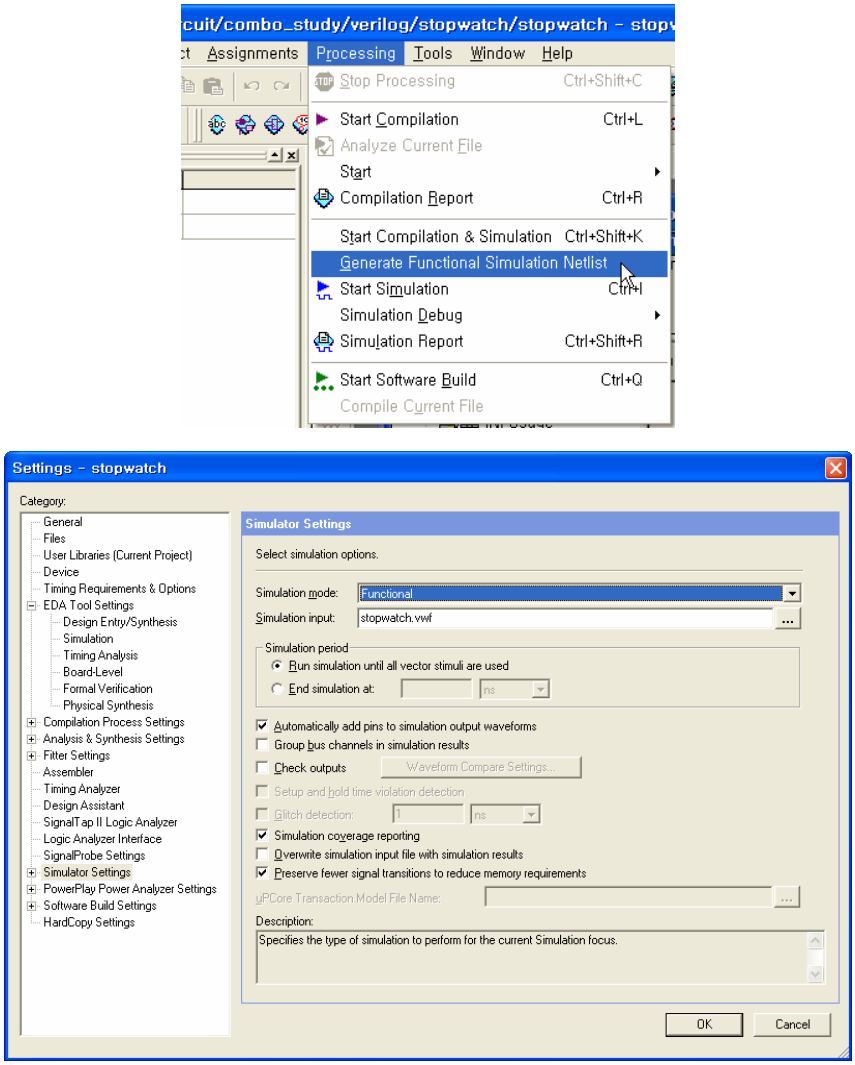
[그림 4-44] Simulation Report

시뮬레이션에서는 이러한 입력파형에 대한 값과 설계 파일의 정보를 보고 결과 값을 보여주게 됩니다. [그림 4-44]에서는 파형의 입력 창과는 달리 Report 구간에 대한 영역이 나타나 있습니다.

따라서 파형적인 분석과 달리 문서적으로 결과에 대한 리포트를 해 주고 있습니다. 또한 이러한 시뮬레이션은 앞서 설명한 것과 같이 타이밍 시뮬레이션을 기본으로 하고 있습니다. 그러므로 사용자가 아무런 설정 없이는 디바이스에 대한 delay 시간을 고려한 시뮬레이션 과정을 수행하게 됩니다.

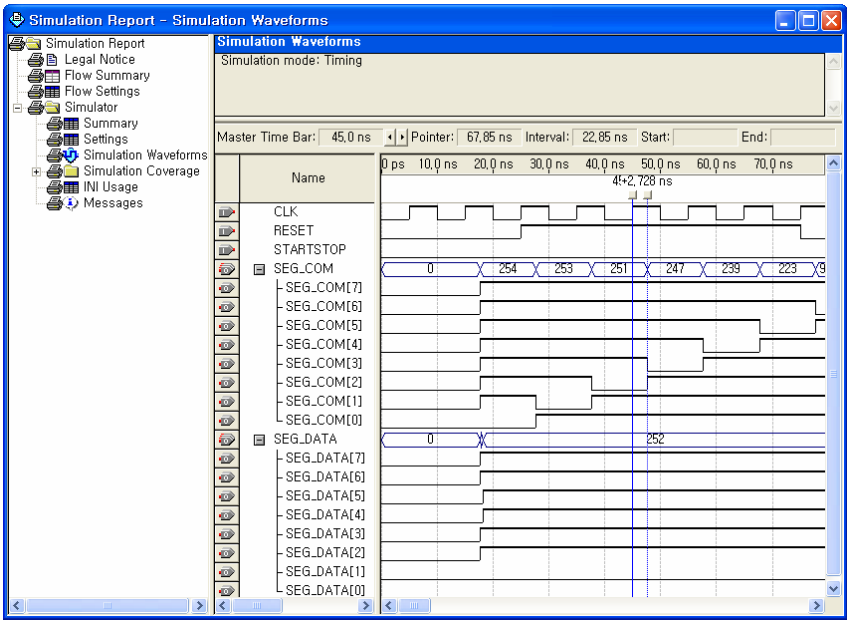
Function 시뮬레이션 작업을 위해서는 [그림 4-45]와 같이 메뉴의 Processing -> Generate Functional Simulation Netlist 작업을 먼저 해 주어야 합니다. 이 작업을 통해 만들어진 데이터 값을 보고 기능적인 시뮬레이션 작업을 수행하게 되는 겁니다. 그리고 Assignments -> Settings에서 왼쪽 Category를 Simulator Setting로 선택해 주고, 나타나는 시뮬레이션 창의 Simulation mode를 Functional로 선택해 서 시뮬레이션을 실행 시키면 됩니다.

이러한 내용은 [그림 4-45]를 통해 확인할 수 있습니다. 초반에 Generate Functional Simulation Netlist 작업이 수행되지 않고 Setting 창에서 Functional만 바꾸어서 실행을 시키면, 시뮬레이션 과정에서 Netlist 파일에 대한 정보가 없어서 에러가 발생하고 시뮬레이션 과정이 중단되게 됩니다.

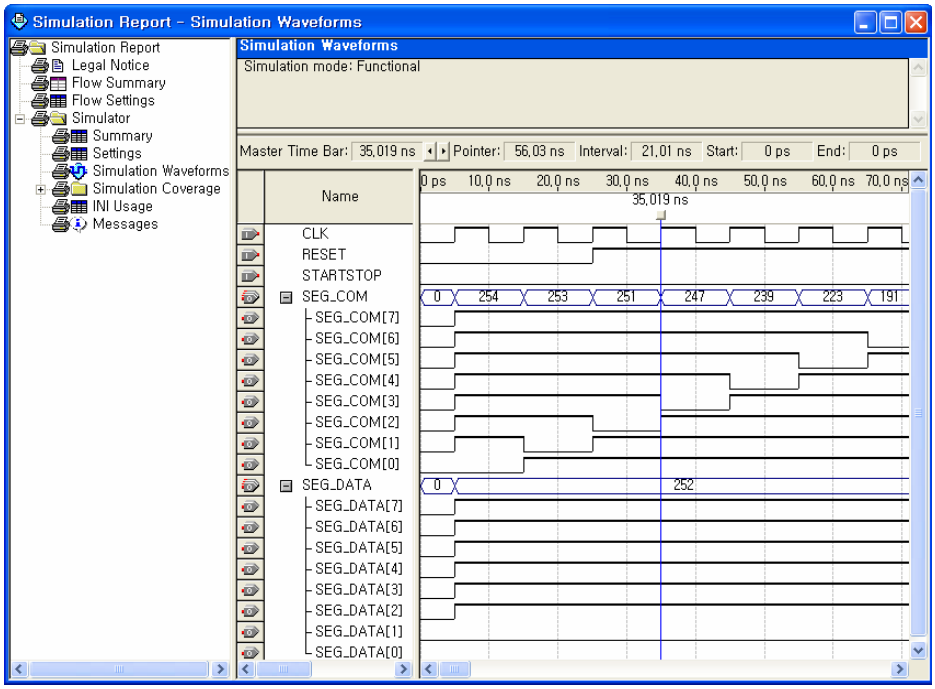


[그림 4-45] Functional Simulation Settings

다음의 [그림 4-46]과 [그림 4-47]은 두 가지의 시뮬레이션 결과를 비교해서 보여주고 있습니다. 타이밍 시뮬레이션에서는 디바이스에서 발생하는 delay 시간을 고려한 시뮬레이션 결과를 보여주고 있습니다.



[그림 4-46] Timing Simulation



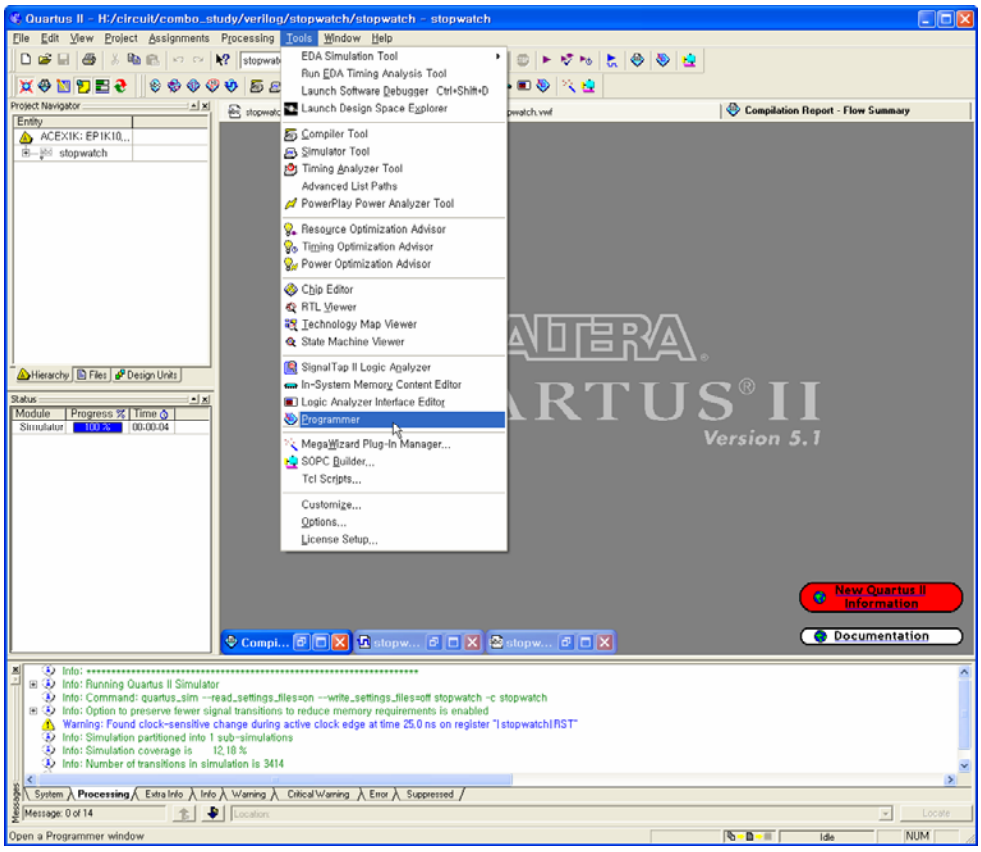
[그림 4-47] Function Simulation

7) Programming

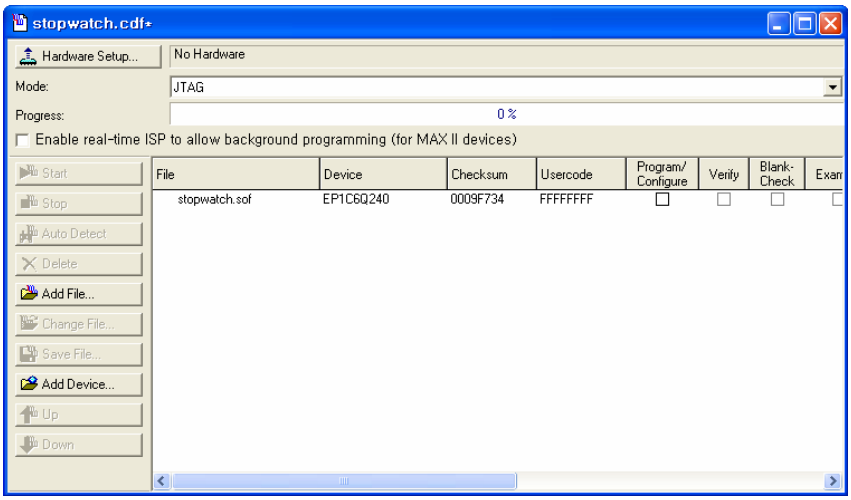
위에서 설계 및 검증이 끝난 논리 회로를 디바이스에 프로그래밍 부분입니다.

① Programmer 활성화

[그림 4-48]과 같이 Quartus II 메뉴에서 Tool –> Programmer에서 Programmer 창을 활성화해서 사용할 수 있습니다. 이 메뉴를 선택하면 [그림 4-49]와 같이 Programmer창이 활성화 되고 여기에서 장비에 직접 다운할 수 있습니다.



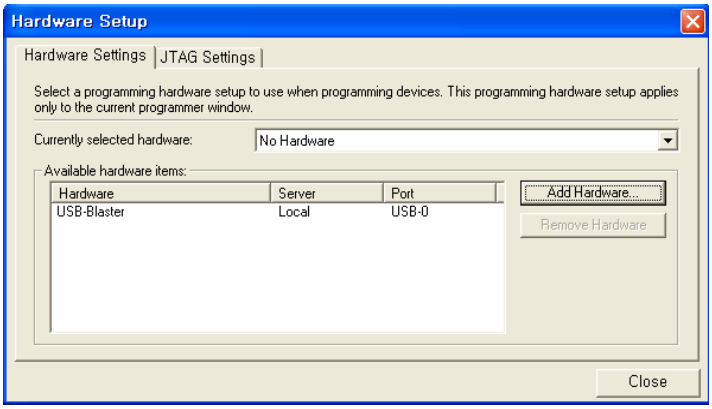
[그림 4-48] Quartus II 메뉴에서 Programmer 항목 선택

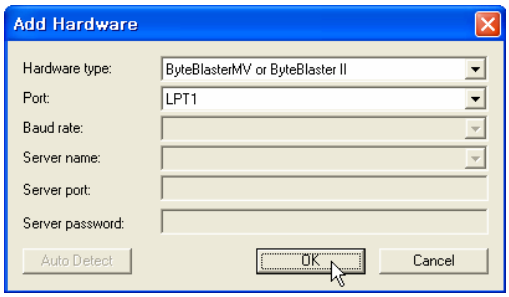


[그림 4-49] Programmer 창 활성화

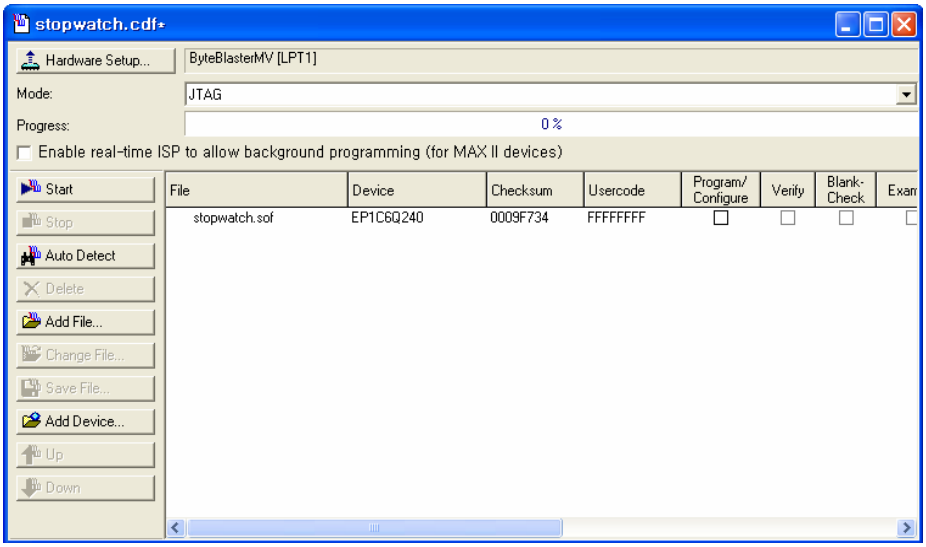
② 하드웨어 설치

처음에 Programmer 창을 활성화 하면 Hardware Setup 창에 No Hardware라는 것이 표시가 됩니다. 여기에서는 ByteBlaster라는 하드웨어를 가지고 Altera device를 프로그램 하기 때문에 이러한 하드웨어를 등록을 해 주어야 합니다. Programmer 창 위에 보이는 Hardware Setup를 클릭하면 하드웨어를 등록해 주는 창이 활성화 됩니다. [그림 4-50]와 같이 Hardware Setup 창에서 ByteBlaster를 등록해 주면, [그림 4-51]과 같이 ByteBlaster의 등록된 모습을 볼 수 있습니다.





[그림 4-50] Hardware Setup

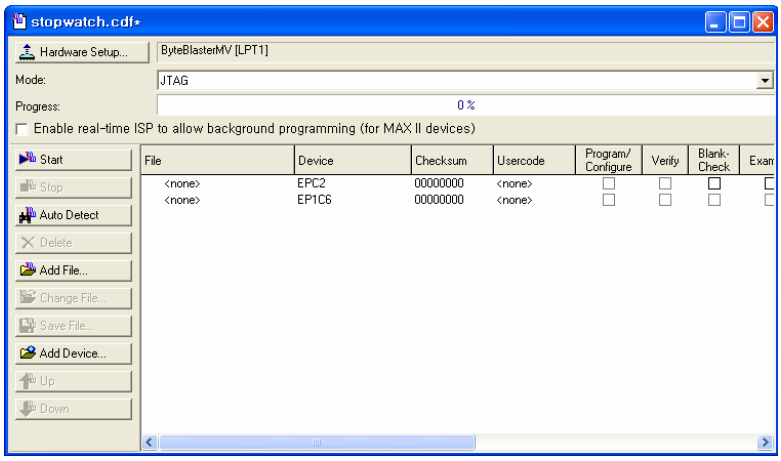


[그림 4-51] Hardware Setup 완료

③ 프로그래밍

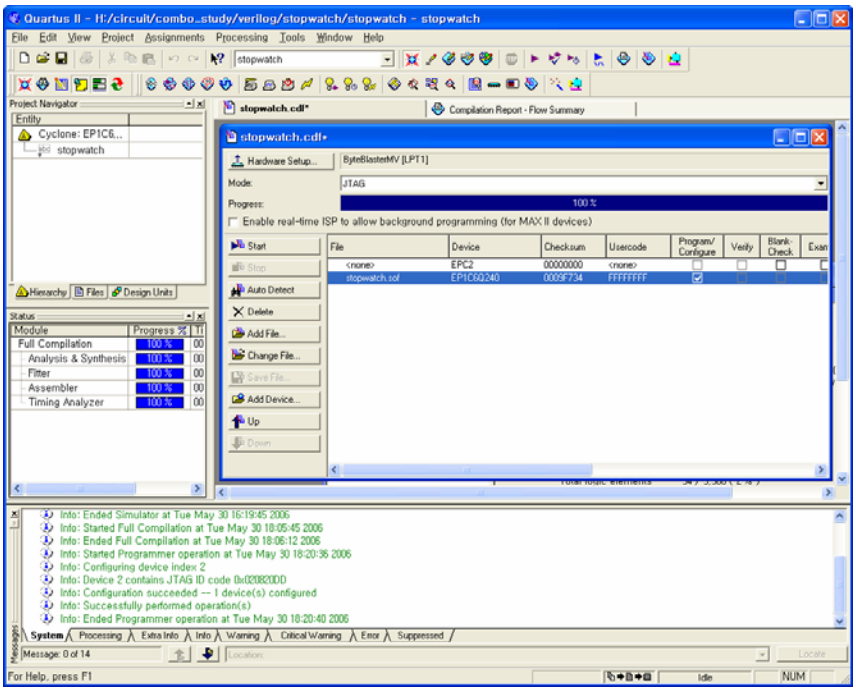
하나의 프로젝트가 선언되고 컴파일을 통해 프로그래밍 파일이 만들어 졌을 때, Programmer 창을 활성을 하면, File에 현재 프로젝트에서 프로그램 할 수 있는 .sof 파일과 device정보 등이 보이게 됩니다. 여기에서 Program/Configure에 대한 부분을 체크하고 Start 버튼을 클릭하여 다운 하면 됩니다.

만약 동작이 되지 않을 때, Programmer 창에서 활성화 되는 부분의 버튼인 Auto Detect 버튼을 눌러 디바이스와 PC간의 연결이 제대로 되었는지 확인해 보기 바랍니다. 따라서 [그림 4-52]과 같이 화면상에 두 개의 디바이스가 나타나는지 확인해 보기 바랍니다. 만약 이 화면과 다르게 표시되거나 “Unable to scan device chain. Can’t scan JTAG chain.”이라는 메시지가 표시되면 연결 케이블에 대한 확인과 디바이스 전원에 대한 확인을 해 주고, 이 사항에 대한 문제가 없을 시 회사로 문의 해 주기 바랍니다.



[그림 4-52] Auto Detect

위 사항에서 아무런 이상이 없이 진행이 되면, 다운로드 케이블을 PC의 Parallel port와 보드의 다운로드 port에 연결한 후 Start 버튼을 누르면 디바이스에 다운이 됩니다. [그림 4-53]에는 다운이 완료된 모습을 보여주고 있습니다. 이렇게 다운이 완료되었을 때, 메시지 창에 Configuration succeeded 메시지가 표시되며 보드에서 설계한 회로가 바로 동작이 진행되게 됩니다.



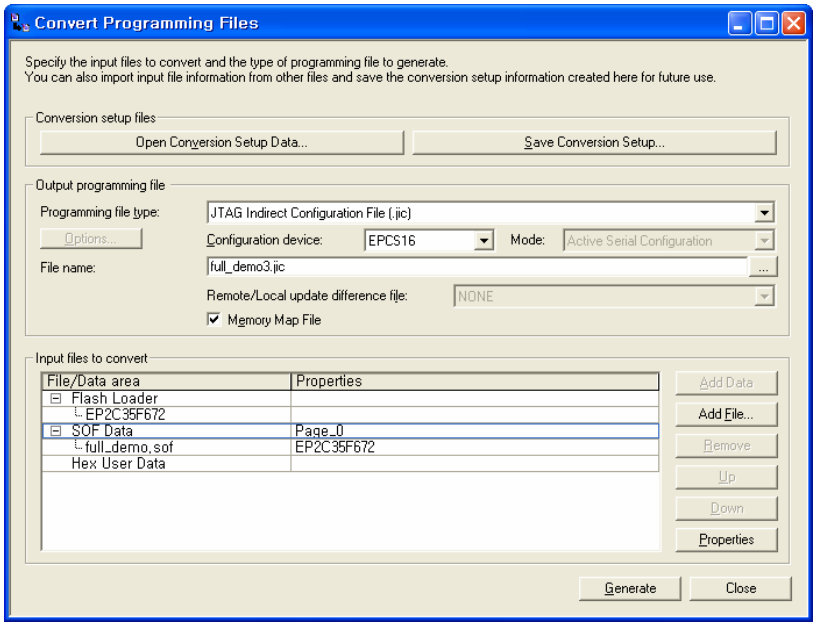
[그림 4-53] Configuration Ended

④ Configuration ROM Programming

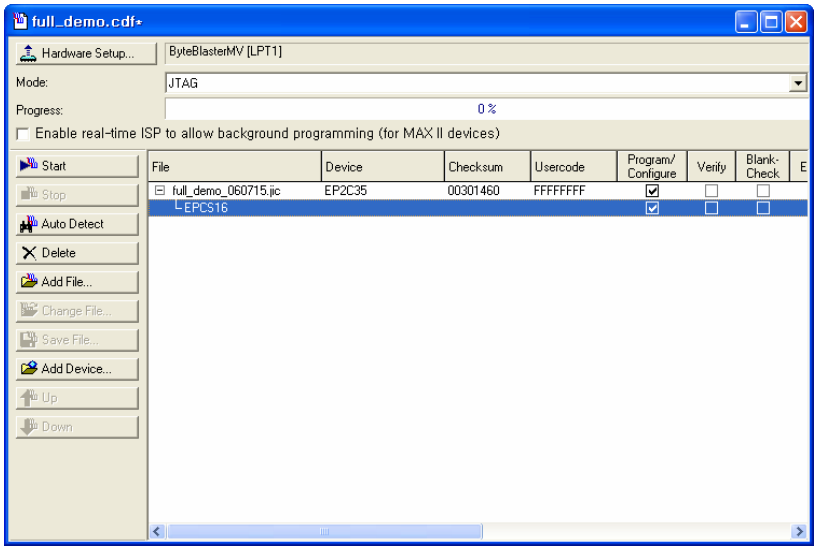
COMBO II 보드에는 Cyclone II device가 RAM 타입의 디바이스이기 때문에 별도의 ROM 타입의 디바이스인 EPCS16 사용하고 있습니다. 따라서 전원의 ON/OFF시마다 디바이스의 데이터가 사라져 다운을 다시 해야 되는 작업을 막기 위해 별도의 ROM을 두어서 전원의 ON/OFF시 마다 데이터를 ROM에서 FPGA디바이스로 전달하여 보드가 작동이 되도록 구성하고 있습니다.

Configuration ROM에 writing하는 방법은 기존의 프로젝트에서 설계된 .sof 파일을 변환하여 사용해 주면 됩니다. 따라서 Quartus II의 메뉴의 File -> Convert Programming Files에서 이 작업을 수행해 주면 됩니다. 여기에서는 Programming file type는 그대로인 “JTAG Indirect Configuration File (.jic)”로 유지해 주고, Configuration device는 보드에서 사용하고 있는 EPCS16을 사용해야 합니다.

또한 File name는 사용자가 만들려는 적당한 이름을 주면 됩니다. 또한 Flash Loader는 사용하는 FPGA 디바이스인 EP2C35F672를 선택해 주고, SOF Data는 현재 프로젝트에서 설계한 결과 파일인 .sof 파일을 넣어 주면 됩니다. 이상으로 모든 것에 대한 설정이 마쳤으면, Generate 버튼을 눌러 EPCS16에 프로그램 할 수 있는 파일을 생성하게 됩니다. [그림 4-54]에는 파이변환에 관한 창을 보여주고 있고, [그림 4-55]에는 이 결과 파일을 가지고 EPCS16 디바이스에 프로그램 하는 과정을 보여주고 있습니다.



[그림 4-54] Convert Programming Files



[그림 4-55] Configuration ROM Programmer

4.2.2 ISE를 이용한 설계

1) 역할

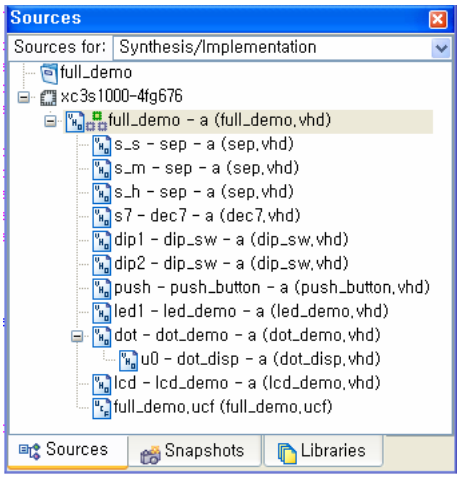
Xilinx ISE 프로그램도 Quartus II와 마찬가지로 프로젝트를 선언하여 프로젝트 중심으로 관리를 하도록 되어 있습니다. 따라서 프로젝트에 속한 설계 파일, 컴파일 관련 파일, 시뮬레이션 파일, 핀 정보 등의 일련의 파일들이 하나의 프로젝트에 속해 관리를 하게 됩니다. 이렇게 구성이 되어 있어, 하나의 프로젝트 파일을 오픈 하였을 때, 프로젝트에 관련된 모든 파일을 불러와서 정보를 보고 사용할 수 있도록 구성을 하고 있습니다.

[그림 4-56]은 VHDL로 작성한 소스 코드를 보여주고 있으며, [그림 4-57]에서는 프로젝트에서 관련된 소스파일들을 보여주고 있습니다.

full_demo.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity full_demo is
5  port(
6     clk         : in std_logic;
7     button_sw   : in std_logic_vector(15 downto 0);
8     dip_switch  : in std_logic_vector(15 downto 0);
9
10    led         : out std_logic_vector(15 downto 0);
11    seg_data : out std_logic_vector(7 downto 0);
12    seg_com     : out std_logic_vector(11 downto 0);
13    lcd_data : out std_logic_vector(7 downto 0);
14    lcd_e    : out std_logic;
15    lcd_rs      : out std_logic;
16    lcd_rw      : out std_logic;
17    dot_data : out std_logic_vector(20 downto 0);
18    dot_scan : out std_logic_vector(14 downto 0);
19    step     : out std_logic_vector(3 downto 0);
20    piezo    : out std_logic);
21 end full_demo;
22
23 architecture a of full_demo is
24
25 component led_demo
26 port(
27    clk   : in std_logic;
28    led   : out std_logic_vector(15 downto 0));
29 end component;
30
31 component dot_demo
32 port(
33    clk      : in std_logic;
34    dot_d : out std_logic_vector(20 downto 0);
35    dot_scan: out std_logic_vector(14 downto 0));
36 end component;
37
38 component lcd_demo
```

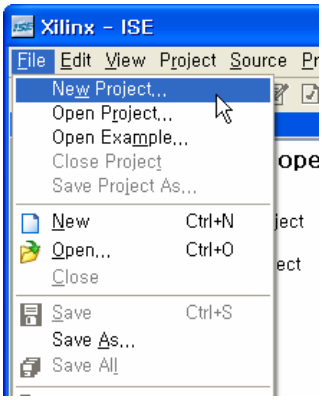
[그림 4-56] full demo 예제



[그림 4-57] Source Files

2) Project 선언

[그림 4-58]와 같이 ISE 프로그램은 File -> New Project 메뉴를 선택하여 프로젝트를 선언할 수 있습니다. 프로젝트 선언은 프로젝트 명 선언에서부터 디바이스 선택에서 파일 추가까지 총 5단계로 이루어져 있는데, 이 과정을 거쳐야 실제 코딩을 위한 프로젝트 선언 과정이 완료됩니다.

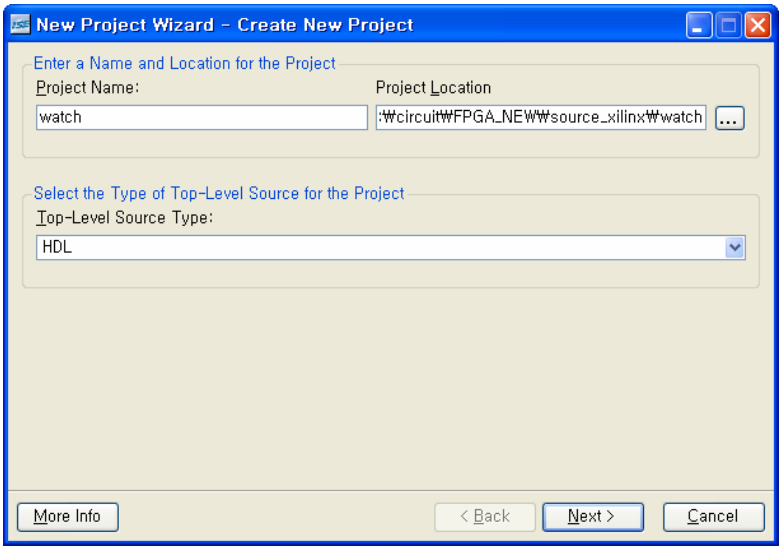


[그림 4-58] New Project

각 단계에서 다음 단계로 진행하기 위해서는 Next 버튼을 클릭해야 합니다. 프로젝트 선언을 시작하면 프로젝트에 대한 전반적인 진행 사항을 설명하는 창이 활성화되고 Next 버튼에 의해 다음 프로젝트 선언을 시작하게 됩니다.

첫 단계로 프로젝트 관련 파일들의 Project Name, Project Location, Top-Level Source Type을 설정 해 주어야 합니다. Project Name을 선택할 때는 Project Location에도 같은 이름

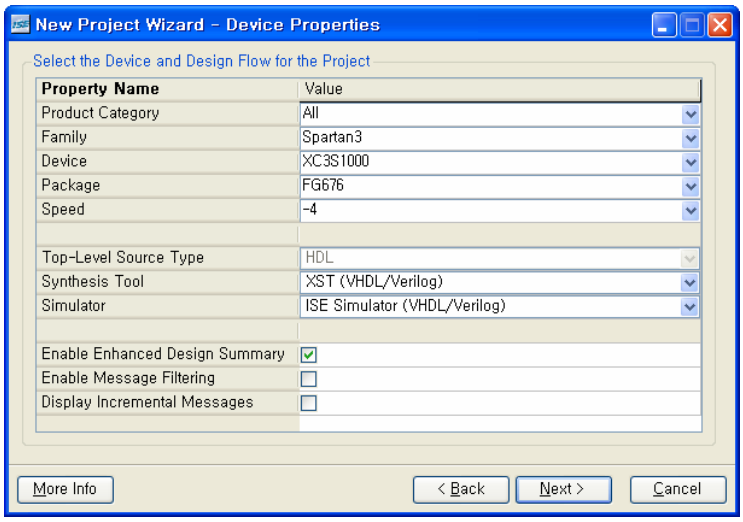
의 폴더가 자동으로 생성하여 그 폴더 내에서 관리 하도록 하고 있습니다. Top-Level Source Type는 현재 프로젝트의 top 파일의 타입을 설정해 주는 것입니다. 여기에는 HDL, Schematic, EDIF, NGC/NGO의 파일 형태를 선택해 주면 됩니다. [그림 4-59]에서는 프로젝트의 이름을 watch 주었고, 이에 관련된 파일들은 C:\watch에 저장되도록 설정했습니다. 또한 top 파일을 VHDL파일 형태로 설계를 할 것이기 때문에 Top-Level Source Type을 HDL로 설정한 것입니다.



[그림 4-59] Create New Project

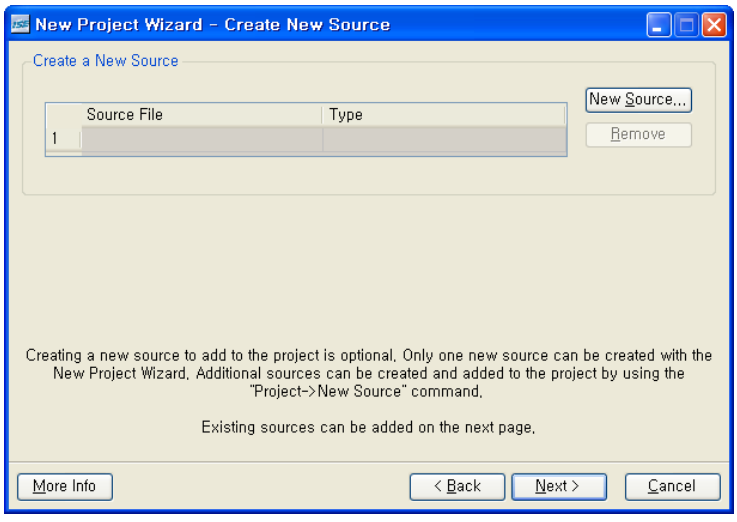
다음은 프로젝트 설정 단계에서 디바이스 설정 및 EDA 툴 설정을 하는 단계입니다. 따라서 현재 사용하는 디바이스를 선택하고 합성과 시뮬레이션 툴을 선택하여 설정해 주는 작업을 하는 단계라고 볼 수 있습니다.

[그림 4-60]에서는 현재 작업을 설정한 모습을 보여주고 있습니다. Spartan3 Family의 100만 게이트 676핀 사양을 가진 디바이스를 선택한 모습을 보여주고 있으며, 합성과 시뮬레이션 툴은 ISE에서 지원해 주는 것을 사용하고 있습니다. 아래 그림에서 이와 같은 설정 모습을 볼 수 있습니다.



[그림 4-60] Device Properties

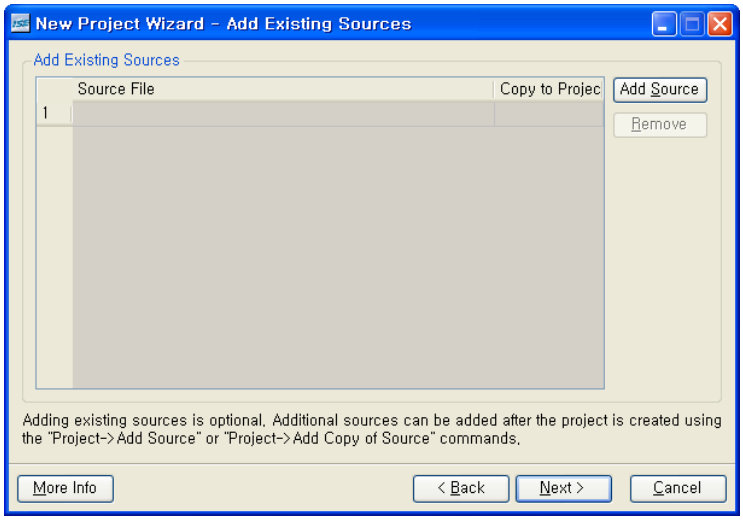
다음 과정은 프로젝트 설정 단계에서 프로젝트에 추가 할 파일 중 새로 생성할 파일의 이름과 타입을 적어주는 단계입니다. 따라서 설계자가 어떤 파일을 생성하여 설계를 할지 미리 생각하고 프로젝트를 선언하고 있다면 이 단계에서 새로 생성할 파일을 적어 주면 됩니다. 새로 생성할 파일을 현 단계에서 넣어 주어도 되고, 프로젝트 선언이 끝난 다음에서도 추가가 가능합니다. [그림 4-61]에서는 Create New Source단계의 창을 보여주고 있습니다.



[그림 4-61] Create New Source

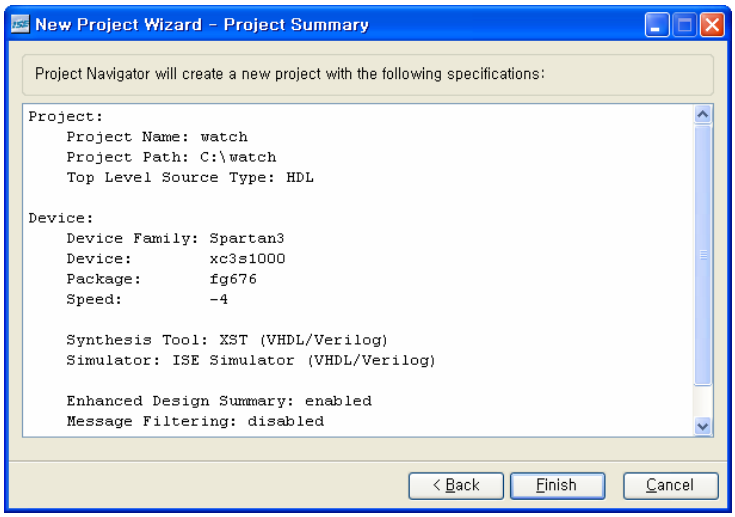
현 단계는 이전 단계와 비슷한 소스 파일을 추가하는 단계이지만 미리 생성한 소스 파일을 추가해 주는 단계입니다. 따라서 사용자가 미리 설계된 상위 또는 하위 파일들

이 있을 시 이것을 현재 생성하려는 프로젝트에 추가해 주는 부분입니다. 이 부분도 프로젝트를 생성한 후에도 설정이 가능합니다. [그림 4-62]에는 Add Existing Sources 단계를 보여주고 있습니다.



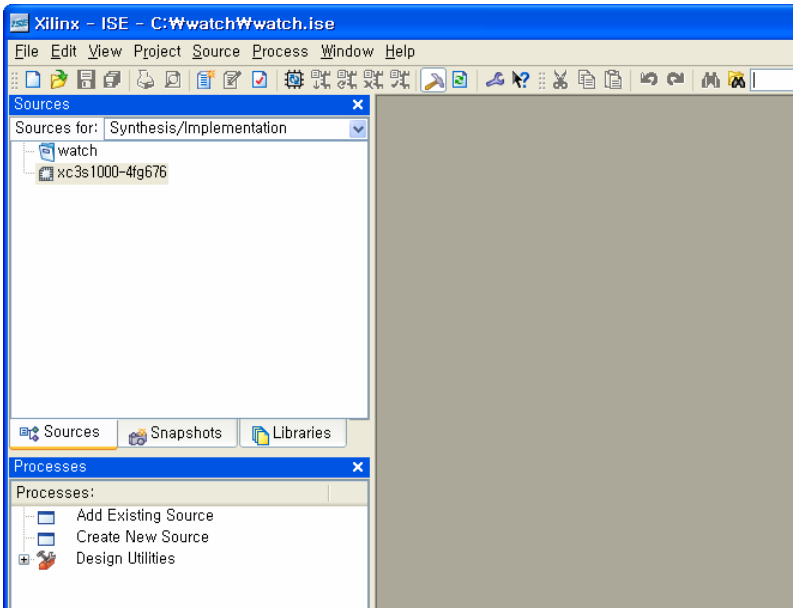
[그림 4-62] Add Existing Sources

이 단계는 프로젝트 설정의 마지막 단계로써 프로젝트 설정 단계에 대한 Summary를 보여주는 창입니다. 따라서 이전 단계에 프로젝트 설정 정보를 다시 확인해 보면서 잘못된 부분이 없는지 다시 확인하는 단계입니다. 현 단계에서 잘 못 설정된 부분을 있으면 Back 버튼에 의해 이전 단계로 이동하여 수정을 하고, 이상이 없으면 Finish 버튼을 눌러 프로젝트 선언을 마치면 되겠습니다.



[그림 4-63] Project Summary

이전까지의 작업으로 현 작업에 대한 프로젝트 선언 작업을 마쳤습니다. 따라서 이 후에 만들어지는 모든 파일들은 현재의 프로젝트 폴더 내에서 되고 관리가 될 것입니다. [그림 4-64]에서는 프로젝트 설정을 마치고 ISE창에 설정된 모습을 보여주고 있습니다. Sources창에서 현재 설정된 프로젝트인 watch와 디바이스인 Spartan3 계열의 디바이스가 보여지고 있습니다.



[그림 4-64] 타이틀에 표시된 프로젝트

3) Design Entry

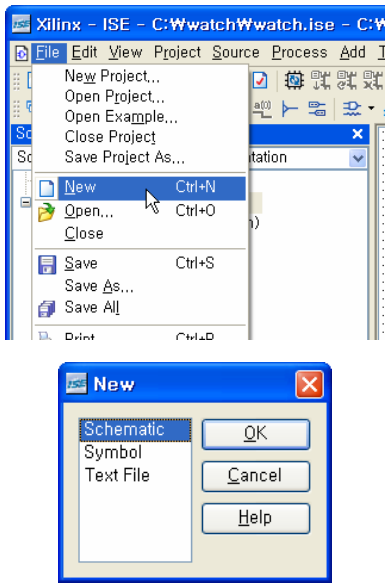
❑ Schematic Editor

① 특징

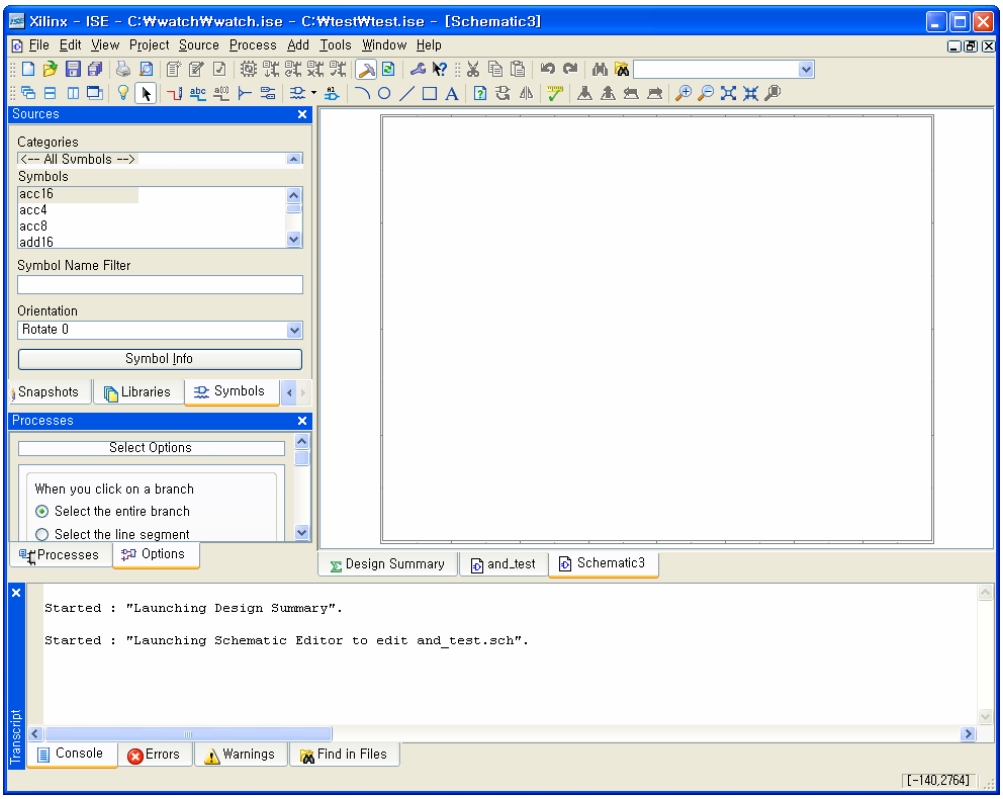
특정 기능을 가진 심볼 라이브러리를 생성하여, 그 심볼 라이브러리의 입 출력 데이터 라인을 선(wire) 등으로 각 심볼 등의 입 출력 데이터 신호를 연결하여 논리 회로를 설계하는 방법입니다.

② Schematic 선택하기

Schematic을 이용해 하나의 논리 회로를 설계하기 위해서는 Schematic Design 창을 활성화 시켜야 하는데, [그림 4-65]과 같이 File -> New 메뉴를 선택하여 New 창을 활성화 한 후 Schematic이라는 항목을 선택하여 OK 버튼을 누르면 [그림 4-66]과 같이 Schematic Editor 창이 활성화 됩니다



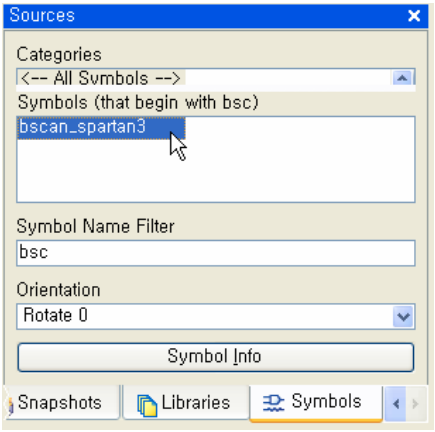
[그림 4-65] 설계 메뉴에서의 Schematic 선택



[그림 4-66] Schematic Editor

③ 심볼 라이브러리

Schematic 으로 작업할 수 있는 환경이 되었으면 이곳에 symbol을 삽입해 주어야 합니다. 따라서 Sources 창에서 아래쪽에 Symbols를 선택하면 아래 [그리 4-67]과 같은 창이 활성화 되고, Symbol Name Filter에서 찾고자 하는 Symbol을 검색하여 찾으면 됩니다. 그림에서는 bscan_spartan3을 선택하는 부분을 보여주고 있습니다. 또한 이 창에서는 Symbol을 회전할 수 있는 Orientation 설정 부분과, 현재 선택한 Symbol에 대한 정보를 확인 할 수 있는 Symbol Info 버튼이 있습니다.

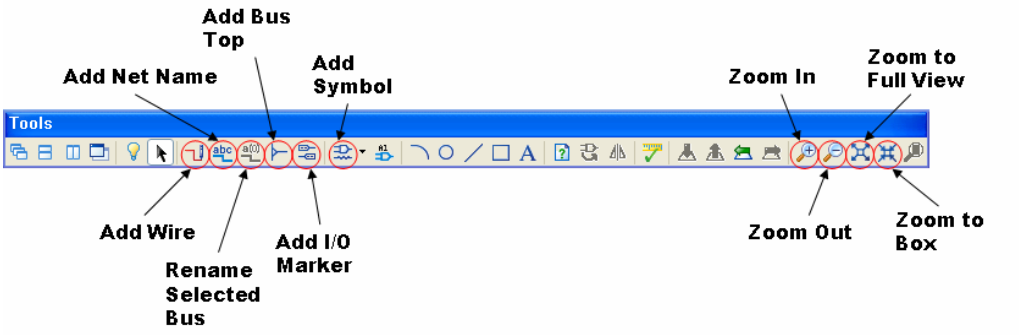


[그림 4-67] Symbols

위의 그림과 같이 심볼의 이름을 알고 있을 때 직접 Symbol의 이름으로 검색하여 불러와 사용이 가능하고 Symbols에서 직접 심볼을 검색하여 찾아 설계 소스에 추가해 주면 됩니다.

④ 각 심볼의 연결

이전의 작업으로 심볼을 불러오면 심볼간의 연결을 해 주어야 합니다. 또한 심볼을 디바이스의 I/O 핀과 연결할 수 있는 포트를 연결해 주어야 합니다. [그림 4-68]는 심볼을 그리는데 사용하는 아이콘들이 나와 있습니다.

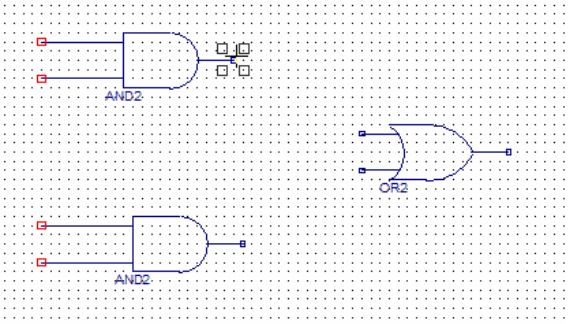


[그림 4-68] Tools

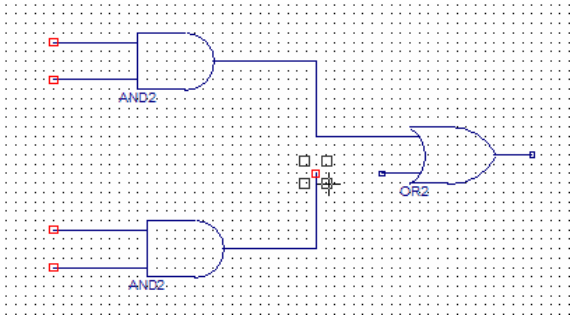
[그림 4-68]와 같이 자주 사용하는 아이콘에서는 Wire를 추가해 주는 부분과, 각 Wire 간의 이름을 주는 아이콘, 심볼을 추가하는 아이콘, 화면에 대한 확대, 축소하는 아이콘들이 포함되어 있습니다. 따라서 설계자가 이러한 아이콘을 통해 Schematic으로 설계를 하는데 쉽게 할 수 있습니다.

[그림 4-69]에서는 심볼들간의 연결하는 모습을 볼 수 있습니다. 왼쪽 상단의 AND2에

마우스를 가져가면, 그림과 같이 사각형 모양의 그림이 4개가 생기게 됩니다. 이것은 심볼의 핀과 연결하여 Wire을 생성하겠다는 표시입니다. 따라서 이 상태로 Wire을 생성하면서 연결하려는 심볼의 핀까지 마우스로 드래그 해 주면 됩니다.

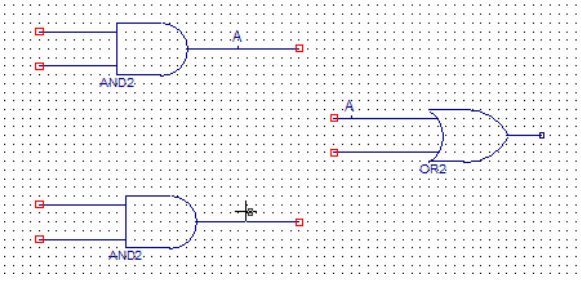


[그림 4-69] 마우스 포인터의 변화



[그림 4-70] Wire 연결

이름으로 연결할 경우에는 연결하고자 하는 Node를 [그림 4-71]와 같이 Symbol에서 wire을 일정 부분 연결한 후 마우스로 이름을 부여하는 모습을 보여주고 있습니다. 메뉴의 Add -> Net Name을 선택한 후에 ISE의 Processes 창에서 Name의 이름을 적어주면 화면의 마우스 커서 옆에 이름이 표시되게 됩니다. 이 상태에서 미리 그려 둔 Wire에 마우스로 클릭하면 Wire에 이름이 표시되게 됩니다.



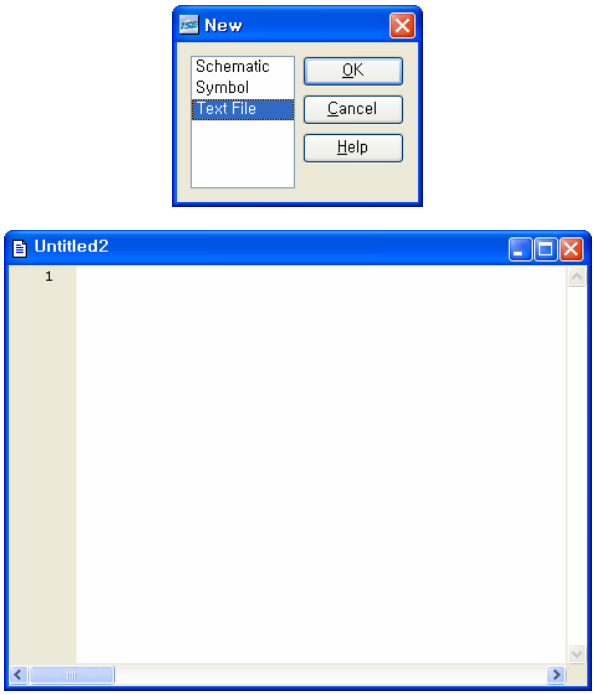
[그림 4-71] 이름으로 연결

❑ Text Editor (HDL에 의한 설계)

HDL에 의한 설계의 기본 내용은 Quartus II에 설명이 되어 있으므로 ISE 설명에서는 생략하고 설계 방법으로 바로 넘어가도록 하겠습니다.

① Text File 선택하기

Text의 HDL을 이용하여 설계하기 위해서는 Text File 창을 활성화 시켜야 하는데, File -> New 메뉴를 선택하면, [그림 4-72]과 같이 어떠한 형태로 설계를 할 것인지 선택하게 된다. 여기에서 Text File를 선택하면 그림과 같이 설계를 할 수 있는 창이 활성화 됩니다. 여기에서 사용자가 HDL을 이용하여 설계를 하고 저장을 하면 됩니다.



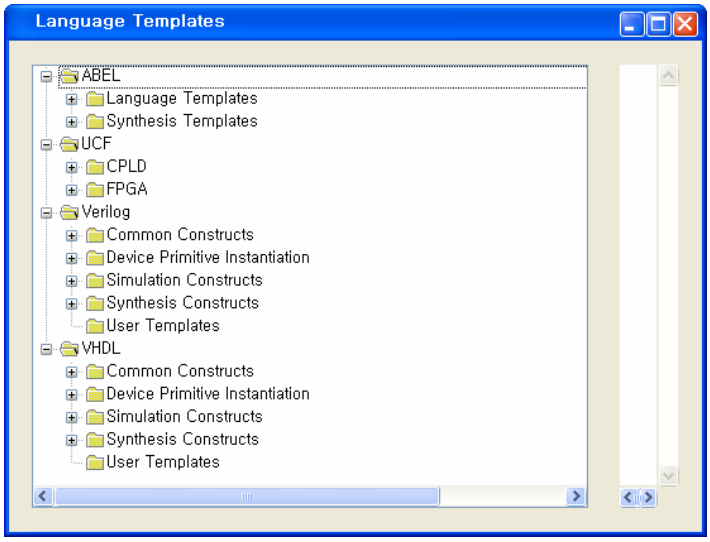
[그림 4-72] Text File

② 설계하기

HDL을 이용한 설계 방법에는 VHDL, Verilog HDL에 의한 설계 방법이 있습니다. 이러한 예제는 Quartus II 툴 설명에서 나온 예제 그림과 같습니다. 저장하는 파일 이름도 Quartus II와 같은 VHDL 파일은 .vhd, Verilog HDL 파일은 .v의 확장자를 가지고 저장합니다.

여기에서도 Quartus II와 마찬가지로 메뉴의 Edit -> Language Templates가 존재하여 각 언

어에 대한 기본적인 현태를 지원해 주고 있습니다. 따라서 사용자는 이러한 메뉴를 이용하여 간단하게 HDL을 작성할 수 있습니다. [그림 4-73]은 Language Templates를 보여주고 있습니다.



[그림 4-73] Language Templates

4) Compile

① 역할

컴파일러는 과정은 Quartus II에서 설명한 같은 목적을 가지고 수행을 하게 됩니다. 여기에서는 Xilinx ISE에서의 컴파일 과정을 살펴보기로 하겠습니다.

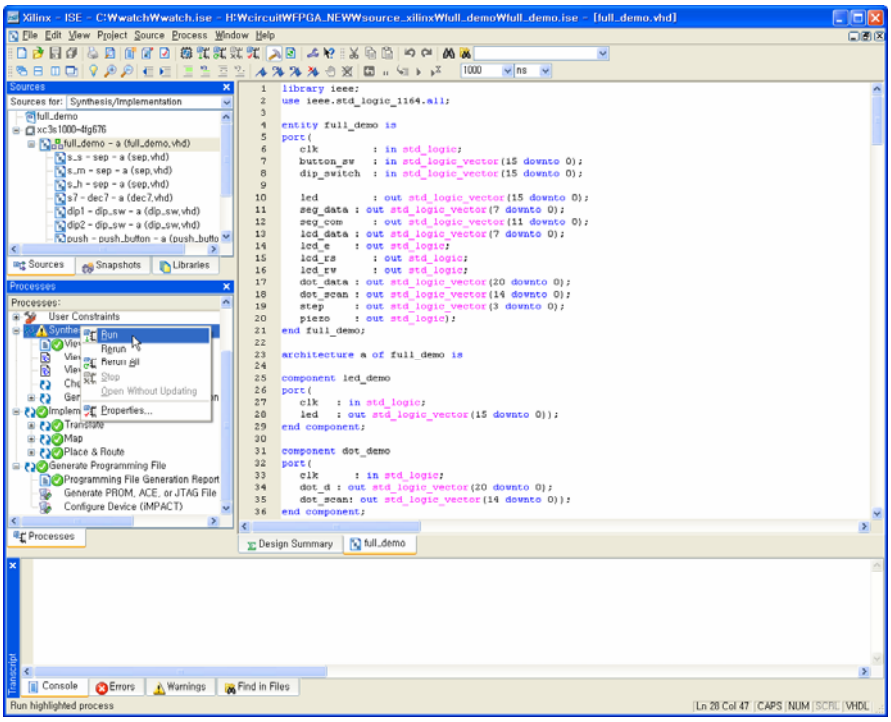
② Compiler

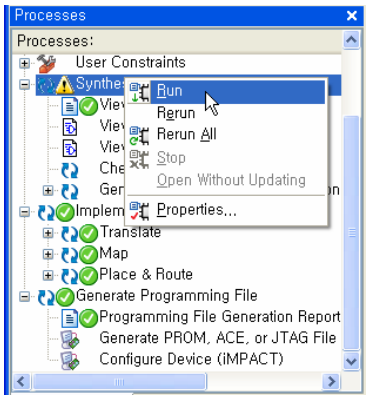
ISE에서는 컴파일 과정을 크게 3부분으로 나누고 있습니다. ISE Processes창에서 볼 수 있듯이 Synthesize, Implement Design, Generate Programming File로 나뉘고 있습니다. Synthesize는 현재 설계한 회로에 대한 부분을 확인하는 작업을 하게 됩니다. 따라서 사용자의 설계 파일을 보고 컴파일을 하는 부분입니다. 설계 소스를 보고 문법적인 오류에 대한 검사를 하고 이에 대한 합성을 하는 단계 입니다. Implement Design은 디바이스를 보고 컴파일 하는 과정입니다.

따라서 디바이스에 대한 디바이스의 로직에 이전에 컴파일 한 소스를 배치하는 단계입니다. 또한 디바이스의 핀에서 어떠한 핀을 사용하여 제어를 할 지 결정하는 단계입니다. 여기에서 설계 소스에서 사용하는 핀을 지정하여 사용할 수 있도록 활성화 시켜주는 작업을 하게 됩니다. Generate Programming File은 프로그램에 관련된 작업을 하게 됩니다. 따라서 프로그램 할 수 있는 파일을 생성하는 작업을 하게 됩니다. 여기에서

는 FPGA 디바이스 및 PROM등의 디바이스에 프로그램 할 수 있는 파일로 변환해 줍니다.

이렇게 나눠져 있는 컴파일 수행 작업은 메뉴의 Process 또는 툴의 Processes 창에서 실행을 할 수 있습니다. 위에 있는 각각의 작업에 마우스 오른쪽 키를 누르면 팝업 창이 활성화되는데, 여기에서 run을 클릭하여 각각의 컴파일 작업을 수행하게 됩니다. [그림 4-74]에서는 이러한 컴파일 작업을 보여주고 있습니다. 여기에서는 Synthesis 작업에 대한 컴파일 수행 작업을 보여주고 있습니다. 나머지도 이 작업과 똑같이 수행해 주면 됩니다.





[그림 4-74] Compile

5) Assignments

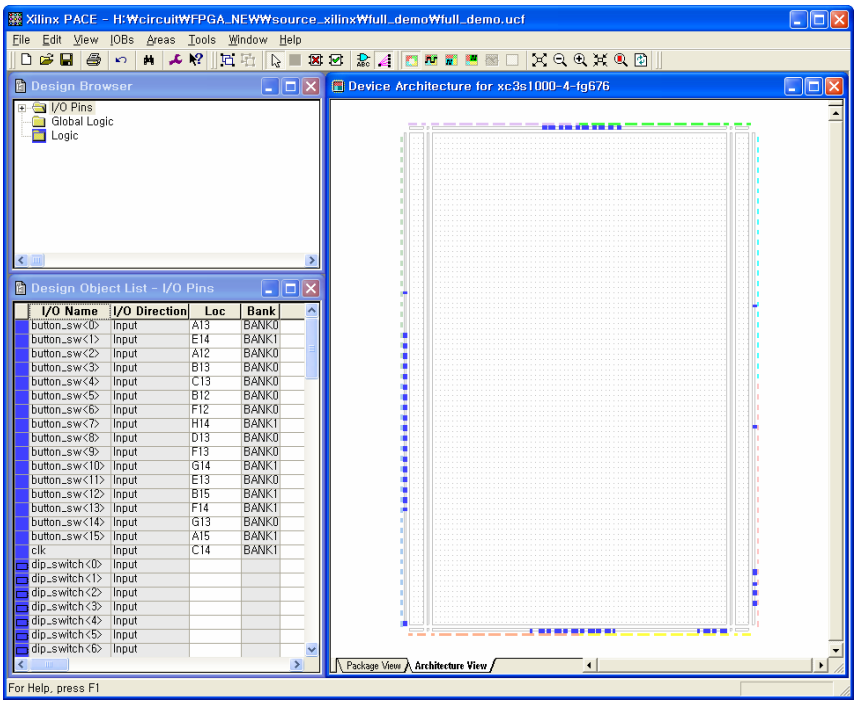
① 역할

이 작업은 디바이스의 I/O핀 설정을 위한 작업입니다. Quartus II에 설명을 참고해 보기 바랍니다.

② Assign Package Pins

프로젝트 설정 단계에서 디바이스 설정을 미리 해 두었기 때문에 여기에서는 핀에 대한 설정만 해 주면 됩니다. 컴파일 과정에서 컴파일러가 포트에 대한 정보를 가지게 됩니다. 따라서 핀 설정 과정에서 사용하려는 핀을 불러와서 타겟으로 사용하는 디바이스에 Assign 하게 되는 것입니다.

이 작업은 Processes 창에서 할 수 있습니다. Processes 메뉴에서 User Constraints -> Assign Package Pins에서 이러한 작업을 수행하게 됩니다. 이 메뉴를 마우스로 클릭하면 확장자가 .ucf 파일을 생성하면서 핀을 할당하는 새로운 창이 활성화 되는데 여기에서 핀을 할당을 할 수 있습니다. 이 창에서는 Design Object List에 이전에 설계한 회로의 I/O Name이 미리 지정이 되어 있습니다. 따라서 사용자는 해당되는 I/O Name에 핀을 할당해 주면 됩니다. Design Object List에서 Loc에 핀 번호를 적어서 할당해 주면 됩니다. [그림 4-75]에서는 핀이 할당된 모습을 보여주고 있습니다.

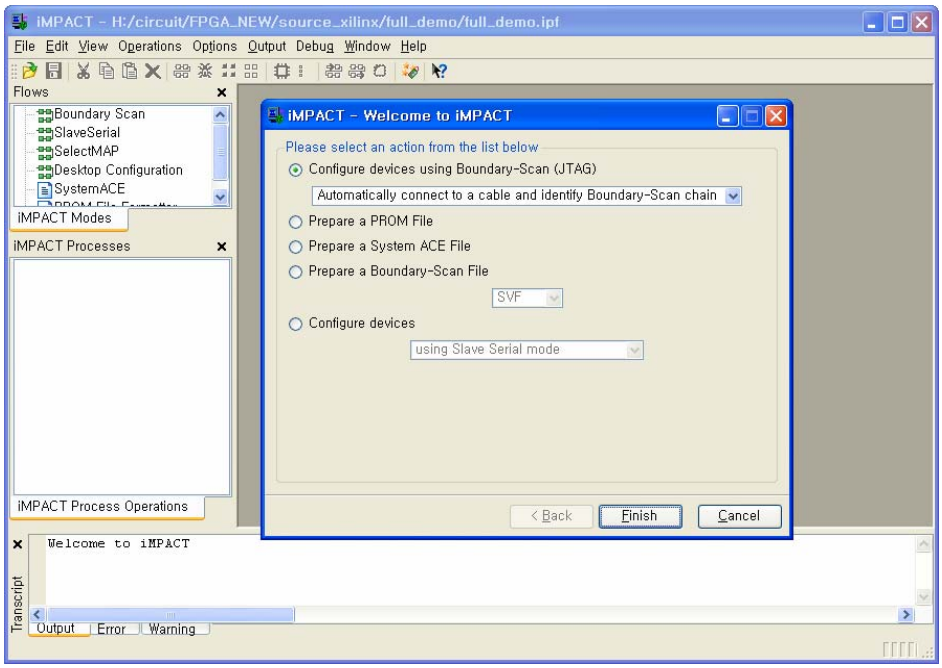


[그림 4-75] Assign Package Pins

6) Configure Device (iMPACT)

Configure Device 작업은 현재까지의 툴에서의 작업을 직접 디바이스에 다운하여 디바이스를 직접 동작해 보는 작업입니다. 따라서 이전까지의 툴에서의 설계 작업을 통한 결과 파일을 디바이스에 직접 다운하여 동작을 시켜 보는 작업이라 할 수 있습니다. ISE에서는 iMPACT를 통해서 이러한 작업을 할 수 있습니다.

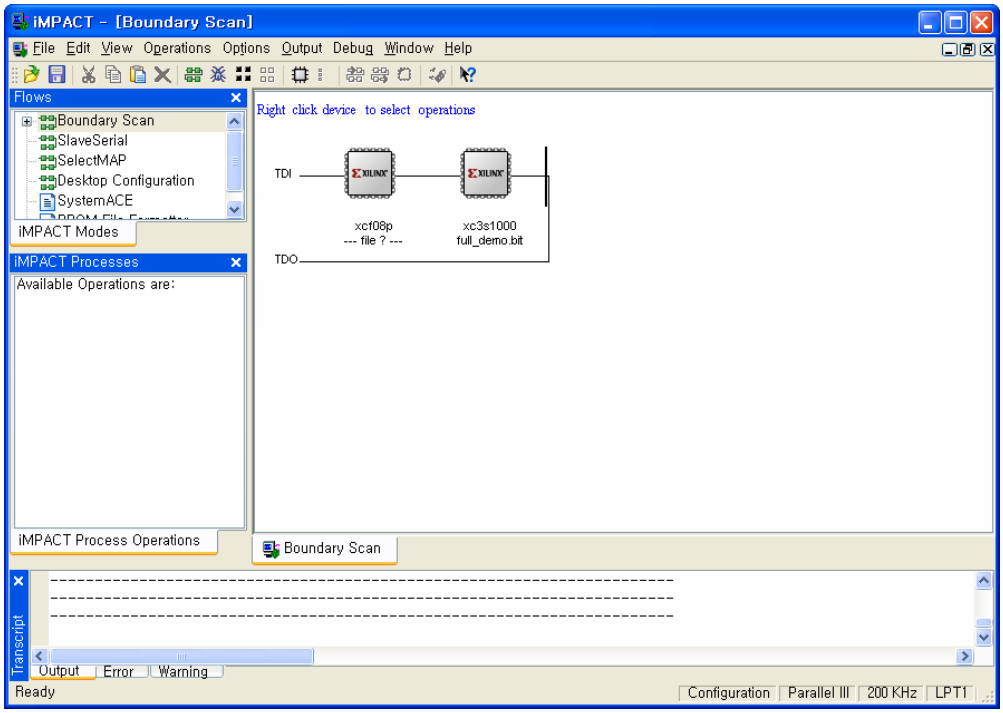
이전까지의 설계와 컴파일 작업이 끝난 상태에서 Generate Programming File를 실행하면 프로그램 할 수 있는 파일이 생성됩니다. 그리고 Generate Programming File -> Configure Device (iMPACT) 를 실행하면 프로그램을 위한 작업 창이 활성화 됩니다. 다음 [그림 4-76]에서는 iMPACT을 실행 시 초기 화면을 보여주고 있습니다.



[그림 4-76] iMPACT

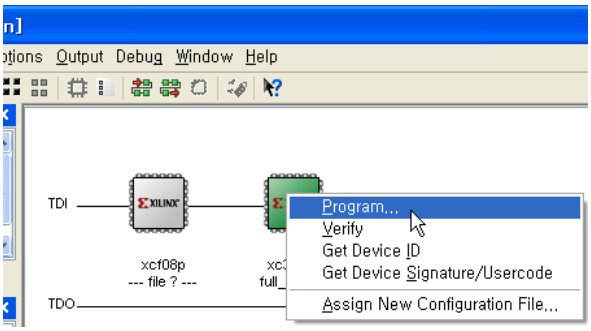
여기에서는 다운하려는 디바이스에 대한 다운 방식을 선택해 주는 작업입니다. 모드와 사용하려는 디바이스가 어떠한 것인지 선택을 해 주어야 합니다. 현재 장비에서는 Spartan3 디바이스를 JTAG 방식으로 다운하는 모드를 취하고 있으므로, 처음에 있는 Configure devices using Boundary-Scan(JTAG)을 선택해서 Finish 버튼을 눌러 주면 됩니다.

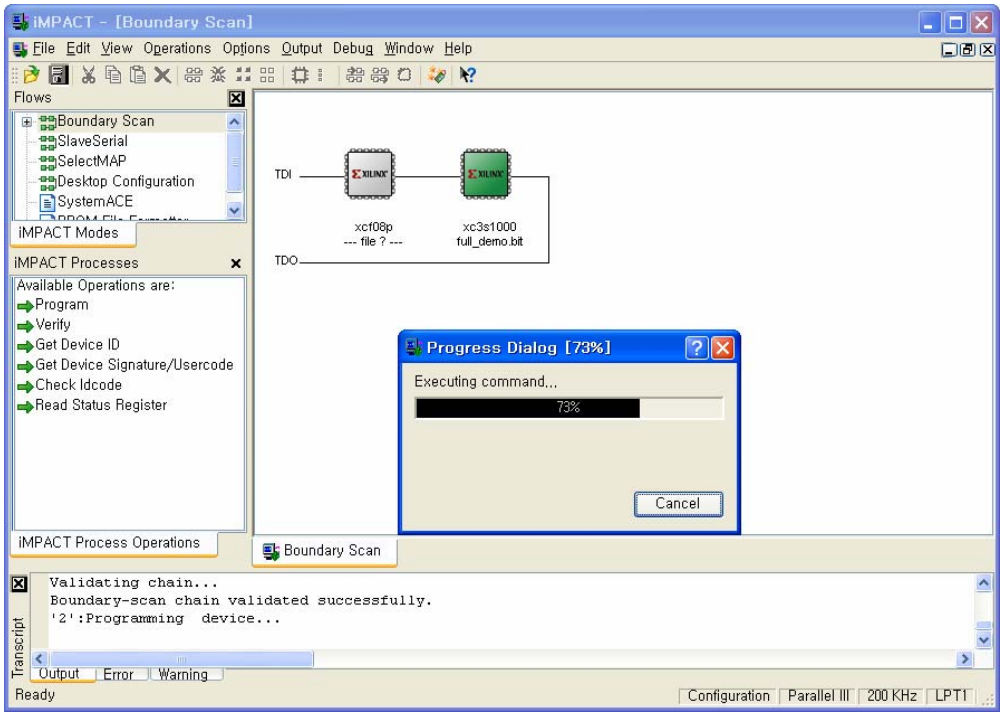
현재 장비에서 사용하고 있는 디바이스는 XC3S1000F676의 FPGA 디바이스와 XCF08의 PROM을 사용하고 있습니다. 따라서 디바이스 검색에서 2개의 디바이스가 검색이 되고, 검색된 디바이스에 어떠한 프로그램 파일을 프로그램 할지 검색할 수 있는 탐색 창이 나타나게 됩니다. 따라서 디바이스 별로 탐색 창이 생성되어 파일을 검색하여 지정해 주면 됩니다. [그림 4-77]에서는 JTAG를 통해 검색된 디바이스와 XC2S1000 디바이스에 프로그램 파일인 full_demo.bit 파일이 지정된 모습을 보여주고 있습니다.



[그림 4-77] iMPACT

이렇게 파일을 지정했으면, 지정된 디바이스에 마우스 오른쪽 키를 눌러 Program 메뉴를 클릭해서 디바이스에 프로그램 하면 됩니다. [그림 4-78]에서 이러한 모습을 보여주고 있습니다.





[그림 4-78] Programming

7) Configuration PROM

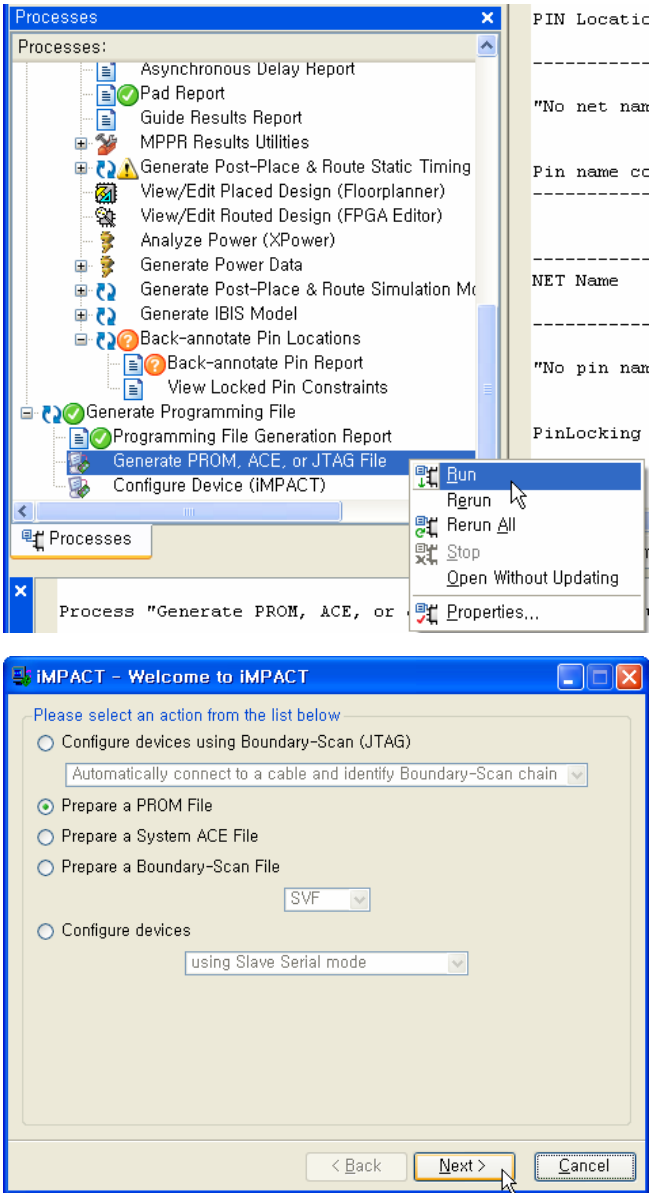
Xilinx 메인 디바이스도 Altera 디바이스와 같이 RAM 타입의 디바이스이기 때문에 별도의 ROM 타입의 디바이스를 같이 사용하여 이러한 점을 보완해 주고 있습니다. 여기에서 사용하는 PROM은 Xilinx사에서 나온 전용 Configuration PROM인 XCF08PVO48을 사용하고 있습니다. 따라서 여기도 마찬가지로 Xilinx사의 tool인 ISE를 가지고 파일의 변환 작업을 수행 하여야 합니다. Spartan 3를 가지고 컴파일 된 최종 파일을 이용하여 변환 작업을 수행합니다.

여기서 주의할 점은 Generate Programming File의 Properties에서 FPGA Start-Up Clock을 CCLK로 해 주어야 합니다. 이러한 작업은 Properties를 활성화 시킨 다음에 Category를 Startup Options을 선택합니다. 그리고 나타나는 하위 메뉴에서 FPGA Start-UP Clock을 CCLK로 설정하고 Generate Programming File을 실행 시킵니다. 따라서 Generate PROM, ACE, or JTAG File을 실행하기 전에 이러한 작업이 완료 되어 있어야 합니다.

이러한 작업은 다운 방식에 따라 어떠한 다운 방식의 클럭을 이용하여 FPGA 디바이스에 프로그램 할 것인지 결정해 주는 작업으로 PROM으로 다운을 할 경우 이러한 작업이 완료가 되어 있어야지만 PROM에 의한 동작이 이루어 질 수 있습니다. 만약 PROM에 다운시 이러한 클럭의 설정 없이 디바이스에 프로그램 하였을 때 PROM에

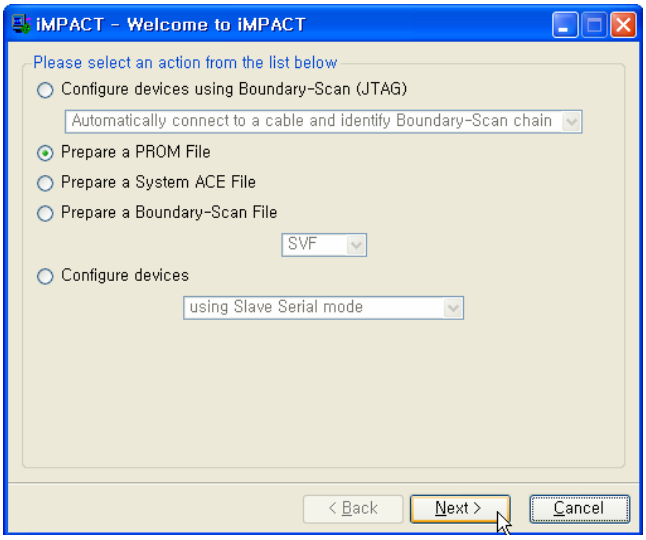
의한 동작이 이루어지지 않을 수 있습니다.

다음의 과정을 통해 이러한 변환 작업을 수행하는 방법이 나와 있습니다. [그림 A-73]은 .bit 파일을 Configuration PROM에 사용할 수 있는 파일로 변환을 위한 작업을 보여주고 있습니다. 따라서 ISE Processes 창에서 컴파일을 수행 후, 위의 그림과 같이 Generate PROM, ACE, or JTAG File를 실행 시킵니다.



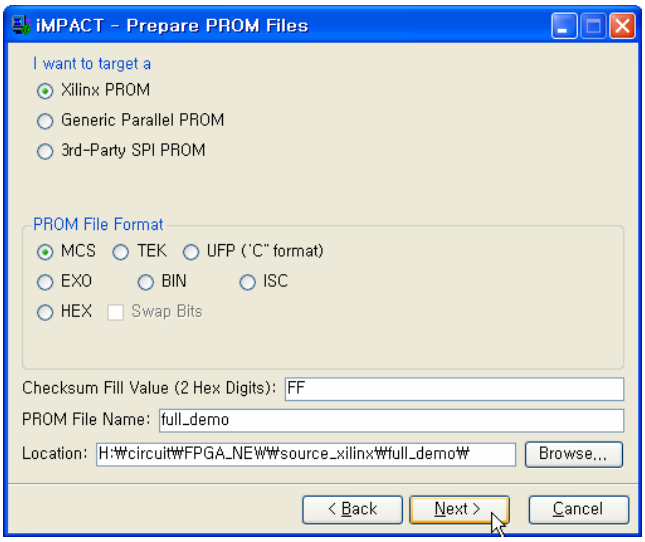
[그림 4-79] Generate PROM, ACE, or JTAG File

[그림 A-74]는 iMPACT를 실행해 디바이스 및 모드 선택을 하는 설정 창입니다. 여기에서는 타겟 디바이스에 관한 프로그램 과정부터 현재 설정을 하려는 PROM 관련 파일 변환 작업을 수행할 수 있는 곳입니다. Xilinx PROM에 적용해 주기 위한 파일 변환을 위한 작업을 수행하려고 하고 있으므로 'Prepare a PROM File' 에 대한 부분을 체크하고 다음 설정 창으로 넘어갑니다.



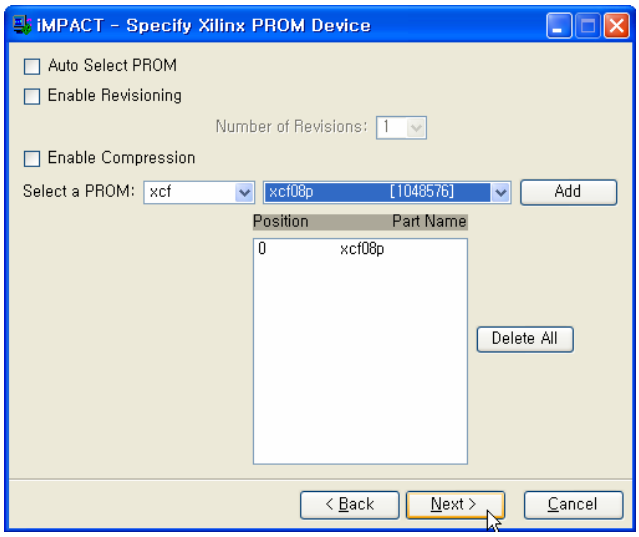
[그림 4-80] Welcome to iMPACT

[그림 A-75]는 생성할 파일에 관한 설정을 해 주는 창입니다. 창에서 윗 부분에 잇는 "I want to targer a ~ " 부분에서 어떠한 종류의 PROM을 사용하고 있는지 선택을 해 줍니다. 여기에서는 현재 장비에 장착되어 있는 'Xilinx PROM' 을 선택해 줍니다. 다음으로 PROM File Format 은 일반적인 PROM 프로그램 파일 형태인 "MCS"를 선택해 줍니다. Checksum Fill Value (2 Hex Digits)는 Default로 설정되어 있는 "FF"로 두고 다음에 있는 파일의 이름과 저장되는 폴더는 사용자가 직접 설정을 해 줍니다. 설정 전의 PROM File Name는 Untitled, Location은 프로젝트가 선언된 폴더로 지정이 되어 있습니다. 다음의 그림에서는 이러한 작업을 수행한 그림이 나와 있습니다.



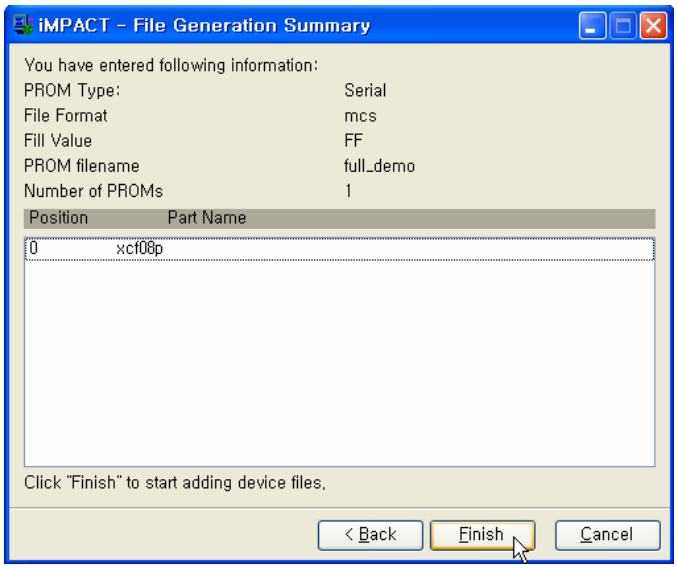
[그림 4-81] Prepare PROM Files

[그림 A-76]은 현재 장비에서 사용하고 있는 Xilinx PROM 디바이스를 선택하는 작업을 합니다. 현재 타겟 장비에서 사용하고 있는 PROM의 종류를 선택해 주고 이러한 PROM이 몇 개 사용하는지를 이 창에서 결정해 주는 것 입니다. 따라서 현재 창에서는 Select a PROM 에서 Xilinx Device Module 에서 사용하고 있는 PROM인 XCF08PVO48 또는 XCF16PVO48을 선택하게 됩니다. 이러한 디바이스를 선택하고 현재 모듈에서 몇 개를 사용하는지를 “Add” 버튼을 이용하여 넣어 주면 됩니다. 여기에서는 현재 디바이스 모듈을 확인해 사용하는 디바이스를 넣어주고 가운데 창을 통해 디바이스 첨가해서 확인을 해 주면 됩니다.



[그림 4-82] Specify PROM Device

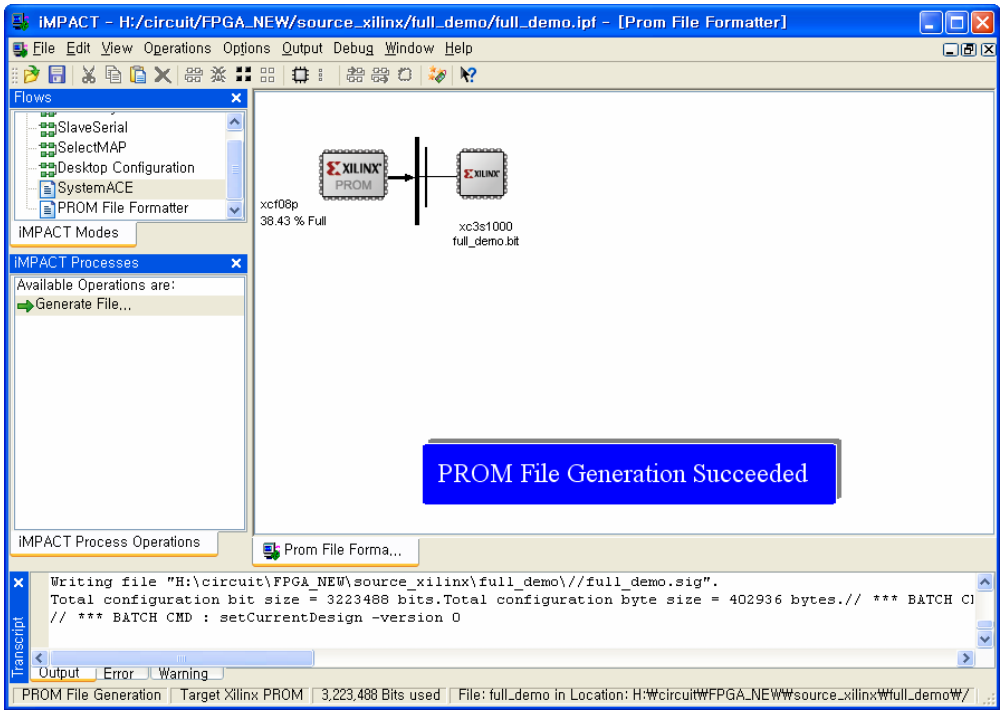
[그림 A-77]은 현재까지의 작업에 대한 Summary 형태로 보여주고 있습니다. 따라서 이전에서 작업한 것에 대한 내용을 이 창을 통해 최종적으로 확인해 볼 수 있습니다. 여기에서 확인한 내용이 이상이 없으면 Finish 버튼을 눌러 마치고, 수정 사항이 이상이 있을 시 Back 버튼을 통해 이 전 단계로 가서 다시 수정을 해 주면 됩니다.



[그림 4-83] File Generation Summary

이상의 작업을 마치고 아래의 Xilinx PROM에 적용을 해줄 파일 변환에 대한 창이 활성화 됩니다. 따라서 여기에서 변환할 .bit 파일을 현재의 작업 폴더에서 찾아서 넣어

주면 됩니다. 이렇게 하고 창에서 마우스 오른쪽 버튼을 이용해 “Generate File...”을 이용해 파일 변환을 수행하면 됩니다. 아래에서 이러한 작업의 완료된 모습을 보여주고 있습니다.



현재까지 두 메인 디바이스에 대한 PROM 파일 변환 작업을 수행하였습니다. 사전에 위와 같이 파일 변환 작업을 하고 Xilinx 디바이스의 전용 다운로드 케이블을 이용하여 프로그램 수행을 하면 됩니다. 프로그램 하는 방법은 이 전의 FPGA 디바이스를 프로그램 할 때 PROM의 파일을 검색해서 포함해 주고, PROM도 같이 프로그램 해 주면 됩니다.

05

HBE-COMBO II
User's Manual
&
Lab Guide

HBE-COMBO II

5. HBE-COMBO II

5.1 기본 구성

본 장비는 다음의 네 부분으로 크게 구분할 수 있습니다. FPGA부, 클럭 제어 부, 내부 장치부, 확장 부로 나누어지며 장비를 사용할 때는 이들을 조작하여 필요한 동작을 위한 준비를 하여야 합니다.

FPGA부는 디바이스를 이용한 직접적인 제어를 하는 곳으로, HBE-COMBO II의 Base board의 커넥터를 통한 FPGA 디바이스를 장착하여 사용이 가능합니다. FPGA 모듈은 Altera, Xilinx 사의 2 종류의 디바이스를 모듈화 하고 있기 때문에, 모듈의 탈 장착으로 두 회사의 디바이스를 모두 사용할 수 있습니다. 따라서 HBE-COMBO II 장비에서는 벤더의 제한 없이 사용이 가능합니다. 각 디바이스 모듈의 구성은 FPGA 디바이스, 각 디바이스의 PROM, 전원, 리셋 단, JTAG port가 구성되어 있습니다. 따라서 커넥터를 통한 전원의 공급만으로 디바이스 모듈의 단독 구동이 가능합니다.

클럭 제어 부는 HBE-COMBO II의 base board에 구성되어 있습니다. 이 부분은 FPGA 디바이스에 공급하는 클럭을 제어하는 부분입니다. FPGA 디바이스는 오실레이터를 통해 클럭을 공급 받아서 사용하는 환경을 가지고 있습니다. 하지만 보드에 장착되어 있는 클럭느 50MHz의 주파수를 가지고 있는 것으로 사용자가 설계 환경에서 사용하려는 클럭은 이러한 주파수 값을 분주하여 사용해서 사용해야 하는 불편함이 있습니다.

따라서 이러한 불편을 해소하기 위해 제어부를 두어 클럭을 분주해서 FPGA 디바이스로 공급해 주는 역할을 하고 있습니다. 이러한 클럭의 공급은 0Hz에서 50MHz까지의 16분주를 Clock Select 스위치를 이용하여 FPGA 디바이스로 공급이 가능합니다. 이러한 클럭의 값은 클럭 제어부의 7-Segment에서 입력되는 주파수를 확인할 수 있습니다. 7-Segment에서 주파수의 숫자를 확인할 수 있고 7-Segment 옆에 있는 3개의 LED를 통해 MHz, KHz, Hz를 확인할 수 있습니다. 이러한 클럭은 Base board의 50MHz의 클럭을 분주하여 디바이스로 공급해준 값입니다.

이 외에 디바이스 모듈에서 별도의 오실레이터를 사용하여 주파수를 공급해 줄 수 있습니다. 이 때, Base board의 주파수는 0Hz로 해야 하고, 디바이스 모듈의 클럭 제어 스

위치를 이용하여 User Clock을 활성화 시켜 주어야 합니다. 이 상태에서 모듈에 딥 타입의 오실레이터를 장착하여 디바이스에 주파수 공급이 가능합니다. 여기에서 사용하는 오실레이터는 Half, Full의 두 가지 타입이 모두 가능하고 사용하는 오실레이터 값이 그대로 FPGA 디바이스로 전달 됩니다.

내부 창치부는 HBE-COMBO II 장비에 구성되어 있는 장치들로, 포트를 이용한 외부의 장비와 통신을 위한 부분과 내부의 장치를 제어하는 부분들이 있습니다. 세부적으로 살펴보면 입 출력 포트로는 먼저 VGA 포트로 모니터 제어를 위한 것과, 2개의 UART 포트를 이용하여 PC등과의 시리얼 통신 실험을 위한 부분이 있습니다. 또한 USB 포트를 이용한 시리얼 통신 실험을 할 수 있습니다. 마지막으로 PS/2 포트를 이용하여 키보드나 마우스를 이용한 제어 실험을 해 볼 수 있습니다.

장비 내부에 있는 실습 장치들로는 Display 장치로 LED, 7-Senment, Dot Matrix등이 있습니다. 또한 VFD(Vacuum, Fluorescent Display)가 있습니다. VFD는 예전의 Text LCD와 제어와 동작은 비슷하지만, 각 dot별 형광 물질에 의한 발광을 하고 있으므로 밝기 면에서 우수한 성능을 보이고 있습니다. 입력 장치로는 키 패드, 버튼 스위치, 버스 스위치의 다양한 입력 스위치를 사용하여 입력으로 사용이 가능합니다. 또한 Piezo를 사용하여 주파수에 따른 음을 조절하여 멜로디 출력을 할 수 있는 부분이 있고, 스테핑 모터를 사용하여 모터 구동 실험을 할 수 있습니다. 마지막으로 적외선 센서를 이용한 통신 실험을 할 수 있는 IrDA 모듈을 사용하고 있습니다.

확장부는 50핀 확장 커넥터 3개를 사용하여 FPGA 디바이스를 디지털 신호를 장비 외부의 장치로 확장해서 사용할 수 있는 부분입니다. 디바이스 모듈 오른 쪽에 있는 EXT 1, EXT 2의 확장 커넥터는 5V의 전원을 포함한 확장이 가능합니다.

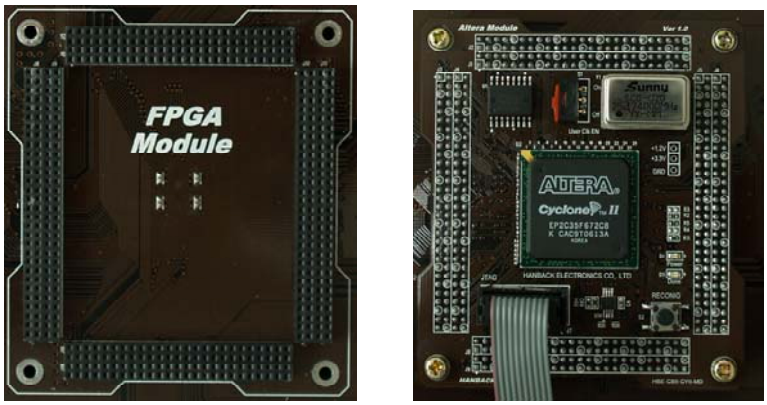
따라서 이러한 확장 커넥터를 통해 한백전자 FPGA Application Module 모두 제어 가능합니다. 이러한 모듈은 신호등, 퍼즐에서 엘리베이터까지 다양한 응용 모듈을 HBE-COMBO II에서 제어가 가능합니다. 그리고 FPGA 디바이스 모듈의 위에 배치하고 있는 EXT 3은 50핀의 확장 커넥터로 5V, 12V의 전원을 공급 받아 사용이 가능합니다. 이 확장 커넥터는 HBE-COMBO II용 확장 모듈을 장착하여 사용할 수 있는 부분입니다. 이러한 보드의 종류로는 신호등, 자판기, ADDA, Stereo Audio Codec Board등이 있습니다. 따라서 이러한 모듈을 HBE-COMBO II 장비에 장착하여 다양한 실험을 해 볼 수 있습니다.

5.2 장비 사용하기

다음은 장비 사용을 위한 순서입니다.

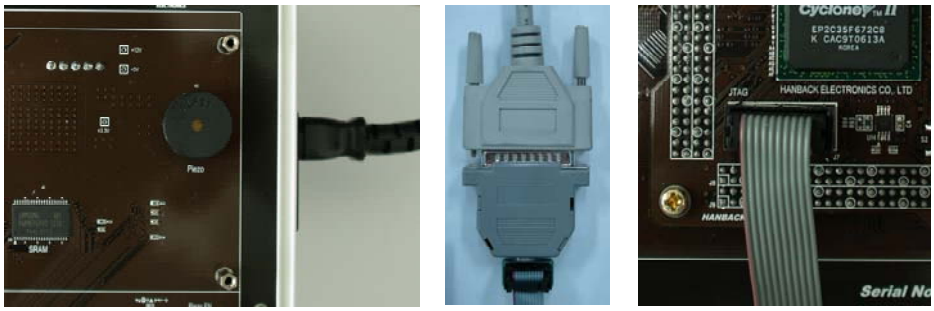
- 본 장비를 활용하기 위해서는 Altera 및 Xilinx 사에서 제공하는 설계 소프트웨어인 Quartus II 및 ISE를 가지고 설계를 하여야 합니다. 또한 모듈에 내장하고 있는 Configuration PROM을 프로그램 하기 위해서도 각 회사의 소프트웨어에서 프로그램 파일 변환 작업을 통해서 PROM에 프로그램 해 주어야 합니다.
- 제품에 포함된 CD-ROM에 각 회사에서 제공되는 소프트웨어 버전의 다운 방법이 포함되어 있습니다. 이 매뉴얼을 통해 다운 받는 방법을 확인 하면 됩니다.

- 사용할 FPGA 모듈을 선택하여 [그림 B-1]과 같이 장착합니다. 장착 시 주의할 사항은 커넥터가 정확히 일치하게 연결되지 않은 상태에서 전원을 인가할 경우 FPGA 디바이스가 손상 될 위험이 있으므로 주의하여야 합니다.
- 또한 디바이스 모듈의 제거 시 한쪽으로만 너무 힘을 가하지 말고 전체적으로 일정한 힘을 가해서 제거 하여야 합니다. 만약 이렇게 하지 않으면 모듈의 제거 시 핀의 손상이 갈 수 있습니다. [그림 5-1]에서는 Altera Module의 장착된 모습을 보여주고 있습니다.



[그림 5-1] Device Module 장착

- 제품에 포함되어 있는 전원 케이블을 제품의 오른 쪽 측면에 있는 전원 연결 포트에 연결 합니다. 이 때 제품의 전원 스위치가 꺼져 있는지 확인한 후 전원 케이블을 전원 플러그에 연결합니다. LPT용 케이블을 PC의 프린트 포트에 연결하고 다른 한쪽을 각 디바이스 별로 있는 JTAG 케이블과 연결해 줍니다.
- 이 JTAG 케이블의 다른 한 쪽을 FPGA 디바이스 모듈의 JTAG 포트와 연결해 줍니다. 이러한 케이블의 연결로 PC에 있는 설계 소프트웨어에서 장비에 있는 FPGA 디바이스를 제어할 수 있습니다. 이 사항은 [그림 5-2]에서 연결된 사항을 확인할 수 있습니다. 여기에서는 Altera Module을 이용하여 연결하는 모습을 보여주고 있습니다.



[그림 5-2] 전원 및 다운로드 케이블 연결

- 장비의 오른 쪽 측면에 있는 전원 스위치를 ON 하여 Base 보드의 Power LED와 디바이스 모듈의 +3.3V LED의 불이 제대로 들어오는지 확인하기 바랍니다. 만약 이러한 LED에 불이 들어오지 않는다면 전원 케이블의 연결이 제대로 되었는지, 장비의 부품에서 핀들간 쇼트(단락)된 부분이 없는지 확인하기 바랍니다.
- 전원이 들어오는 초기에는 디바이스에서 나오는 잔류 전류의 영향으로 LED의 불이 들어오는 경우가 있습니다. 이러한 현상은 장비의 고장이 아니고, 프로그램을 하지 않았을 때의 디바이스의 잔류 전류로 인해 이러한 현상이 일어나게 됩니다. 따라서 케이블이나 PROM을 통한 디바이스 프로그램 후 동작이 시작될 때 이러한 현상이 발생하지 않습니다.
- 전원이 정상적으로 들어 올 경우 제품 오른쪽 아래에 있는 Clock Select 스위치를 돌려 원하는 주파수를 선택합니다. 선택된 값은 Main Clock의 7-Segment 창에 숫자에 대한 표시가 되고, MHz, KHz, Hz라는 3개의 LED를 사용하여 주파수대를 확인 할 수 있습니다. Clock Select 스위치는 아래의 표에서 어떠한 클럭으로 조절 되는지 확인 할 수 있습니다.

Clock SW	클럭 입력	Clock SW	클럭 입력	Clock SW	클럭 입력	Clock SW	클럭 입력
0	0Hz	4	100Hz	8	10kHz	C	1MHz
1	1Hz	5	500Hz	9	50kHz	D	5MHz
2	10Hz	6	1kHz	A	100kHz	E	25MHz
3	50Hz	7	5kHz	B	500kHz	F	50MHz

- Clock Select 스위치는 위의 표에서 볼 수 있듯이 0Hz에서 50MHz까지의 16의 클럭을 Base board에서 FPGA 디바이스 모듈로 공급해 줍니다. 만약 디바이스 모듈에 있는 User Clock을 사용할 경우 Base board의 Clock Select 스위치를 0으로 하고 디바이스 모듈에 있는 User Clock 스위치를 ON 하여 사용하시면 됩니다.

- 장비의 구동은 FPGA 모듈의 PROM에 미리 작성해 놓은 데모 소스가 들어 있으므로 전원 스위치를 ON 함과 동시에 데이터를 다운 받아 데모 동작이 되게 됩니다. 만약 사용 중 다른 동작을 원할 때는 PROM을 프로그램 한 후 RESET 버튼을 눌러 새로운 데모 동작을 실행 할 수 있습니다.

※ 각 부분에 대한 자세한 설명은 2장 및 3장을 참조하기 바랍니다.

COMBOII를 이용한 실험 실습

6. COMBO II를 이용한 실험실습

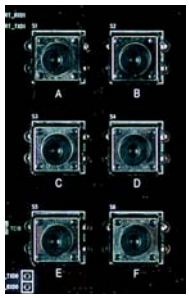
6.1 실험 실습 개요

본 교재에서 권하는 Digital 논리 회로에 대한 실험 순서는 다음과 같습니다.

- Graphic Design(심볼에 의한 논리 회로 설계)으로 논리 회로를 설계합니다.
- 기능적인 시뮬레이션(Functional Simulation)으로 소프트웨어 상에서 설계한 논리 회로에 입력 조건을 주었을 때, 결과가 잘 나오는 지 확인합니다.
- 위의 설계된 논리 회로에 대해 VHDL로 설계한 논리 회로와 비교하여 VHDL로 설계하는 방법을 익힙니다.
- 간단한 논리 회로 또는 조합 논리 회로를 VHDL로 설계한 경우 Graphic Design등의 심볼을 이용하여 설계하는 방법보다 더 복잡하게 느껴질 수 있습니다. 하지만, 복잡한 논리 회로 등을 설계하는데 분석 및 오류 등을 더 쉽게 파악할 수 있기 때문에 최근에는 대부분 VHDL등의 언어를 이용하여 논리 회로를 설계합니다.
- 설계한 논리 회로를 HBE-COMBO II 보드에 프로그래밍하여 보드에 장착된 여러 입출력 장비를 이용해서 위에서 나온 시뮬레이션 결과를 확인합니다.

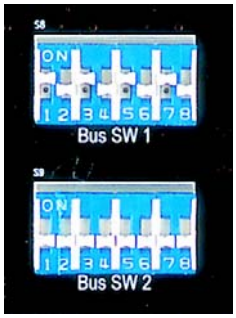
교재에서 연결 소자의 이름과 보드에서의 연결에 대한 설명은 다음과 같습니다.

- 연결 소자의 이름 중 "SW_□"는 버튼 스위치를 나타냅니다. 여기에서 □는 {A, B, C, D, E, F}중 하나를 나타냅니다. 예를 들어 SW_A를 누르면, 버튼 스위치 중 "A" 문자의 버튼 스위치를 나타냅니다. 버튼 스위치를 누르면 High('1')의 값이 입력되고, 버튼 스위치를 누르지 않았을 때는 Low('0')의 값이 입력됩니다.



[그림 6-1] Button S/W

- 연결 소자의 이름 중 "SW□"는 버스 스위치의 입력을 나타냅니다. 여기에서 □는 0에서 15까지의 숫자를 갖고 있습니다. 예를 들어 연결 소자의 이름이 SW0라면 아래 그림의 DIP 스위치 중 첫 번째 DIP 스위치를 말하는 것입니다.
- 버스 스위치로는 DIP Type의 스위치를 사용하는데 각 스위치를 위로 올렸을 때, High('1')의 값이 전달되고, 아래로 내렸을 때는 Low('0')의 값이 전달됩니다.



[그림 6-2] Bus 스위치 입력

- 연결 소자의 이름 중 "LED□"는 연산된 결과를 확인해 볼 수 있는 LED의 출력을 나타냅니다. 여기에서 □는 1에서 8까지의 숫자를 갖고 있습니다. 예를 들어 연결 소자의 이름이 LED6라면 아래 그림에서 왼쪽에서 6번째의 LED를 말하는 것입니다.
- LED에 High('1')의 신호가 전달되었을 때 불이 켜집니다.

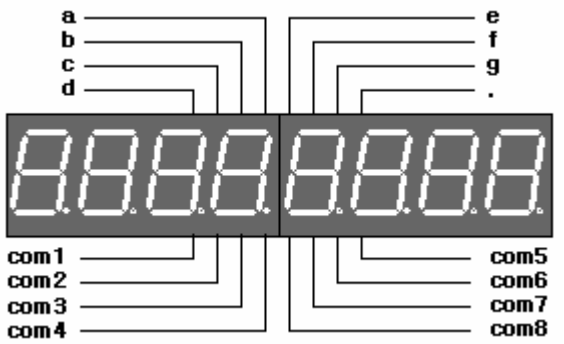
[그림 6-3] LED 출력

- 연결 소자의 이름 중 "SEG_□"는 연산 결과를 확인해 볼 수 있는 7-Segment의 출력을 나타냅니다. 여기에서 □는 A, B, C, D, E, F, G, H의 Segment 데이터 선을 나타냅니다. 예를 들어 연결 소자의 이름이 SEG_B라면 7-Segment의 데이터라인 중 B 부분을 나타냅니다.
- 7-Segment의 데이터 라인 SEG_A에서 SEG_H까지에 각각 '1', '0', '1', '1', '1', '1', '1', '0'의 신호가

전달되었을 때 7-Segment에 60이라는 숫자가 표시됩니다.

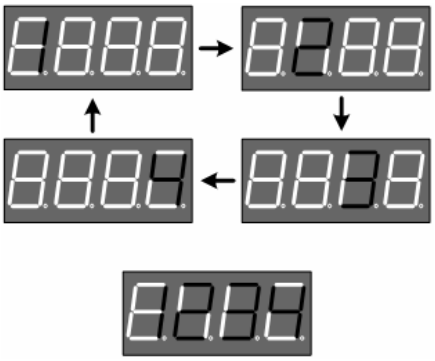
[그림 6-4] 7-Segment 출력

- 본 장비에서는 4개의 7-Segment가 하나로 구성되어 있는 7-Segment LED Array를 두 개 연결하여 아래 그림과 같은 구성되어 있습니다. 7-Segment의 데이터 핀은 8개의 7-Segment에서 공통으로 사용하고 있고, 각각의 common핀을 따로 두어서 8개의 7-Segment를 따로 선택하여 데이터를 출력 시킬 수 있습니다.
- 각각 같은 기능의 핀끼리 연결되어 있고, common 단자도 각각 출력되어 이 common단자를 driving하여 7-Segment에 동작시켜야 합니다.



[그림 6-5] 7-Segment 구성

- 동작 시키기 위해서는 일반적인 방법에서의 7-Segment라면 8개를 사용할 경우 하나의 7-Segment당 출력인 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'가 각각 출력 핀을 가져 64개의 출력 핀이 필요로 하게 됩니다. 이럴 경우 너무 많은 출력 핀이 필요하므로 이를 개선하여 위와 같이 7-Segment의 출력은 공통으로 연결되어 있고 출력할 위치를 지정하는 com1 ~ com8의 값을 제어하여 원하는 숫자나 문자를 표시하는 방법을 사용하게 됩니다.
- 아래의 그림은 6개의 7-Segment에 "1234"의 숫자를 표시하기 위한 방법을 설명한 것입니다.



[그림 6-6] 7-Segment 동작 원리

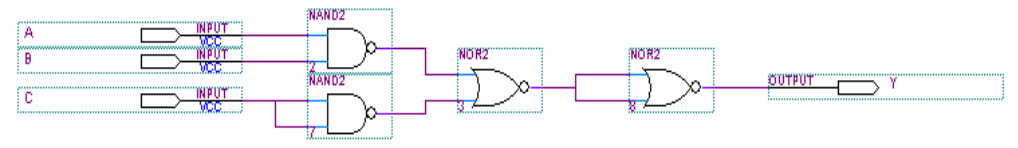
- 다시 설명하면 7-segment 데이터 값을 “00000110”로 주어 ‘1’를 표시하는 값을 주고 com1에 ‘0’을 그리고 나머지 com2~4는 ‘1’의 값을 주면 첫 번째 7-Segment에 1이 표시됩니다. 다음으로 데이터에 “01011011”을 주어 ‘2’의 값을 주고 com2에 ‘0’과 com1, com3~4에 ‘1’을 주면 두 번째 7-Segment에 2가 표시됩니다.
- 이런 순서로 4까지의 숫자를 표시하고 다시 처음으로 돌아가 위의 내용을 반복합니다. 이를 약 1m/s 이상의 주기로 반복하면 눈의 잔상효과에 의해 “1234”의 숫자가 표시되게 됩니다.
- 이 교재에서는 이 driving 하는 부분을 제외하고, 한 7-Segment에서만 데이터가 출력되는 예제를 사용합니다.

6.2 논리 게이트 회로의 응용

6.2.1 실습 1 : NAND 게이트 입력 수의 확장

1) Block Diagram/Schematic File로 설계

❑ 현재 사용하고 있는 설계 소프트웨어의 설계 창에서 심볼을 불러와 [그림 6-7]과 같이 설계를 합니다. 그래픽 에디터로 설계하는 방법은 이전에 설명한 설계 소프트웨어 기본 사용법을 참고해 설계를 하면 됩니다.



[그림 6-7] Graphic Design으로 회로 설계

❑ 설계한 논리 회로를 저장합니다. 저장할 파일 명은 프로젝트 명과 동일하게 하면 되고, 설계한 파일의 확장자는 Altera Quartus II는 .BDF, Xilinx ISE는 .SCH로 저장이 됩니다.

※ 참고 : 프로젝트 명이 EX_1_1이기 때문에 초기값으로 EX_1_1.BDF 또는 EX_1_1.SCH의 이름을 갖습니다.

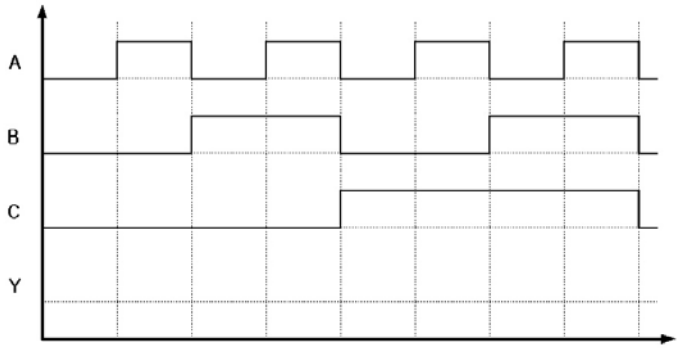
2) 컴파일 및 시뮬레이션을 통한 검증

❑ 설계를 마치면 컴파일 과정을 통해 현재 설계 파일의 문법적인 오류 검사 및 현재의 설계 파일을 분할 합성하는 과정을 거치면서 최종적으로 디바이스에 프로그램 할 수 있는 파일을 생성하고 있습니다.

❑ 따라서 컴파일 과정은 하나의 설계 파일을 분석해 현재의 프로젝트에 정보를 구성하고 최종적으로 디바이스에 프로그램 할 수 있는 파일을 생성하는 과정이라 할 수 있습니다. 이러한 컴파일 과정은 각 회사의 소프트웨어 설명을 참고하여 실행하면 됩니다.

❑ 컴파일 과정이 끝나면 현재 설계 파일을 설계 소프트웨어에서 검증해 볼 수 있는 시뮬레이션 과정을 하게 됩니다. 이러한 과정을 거쳐 현재 설계한 파일이 이상 없이 설계가 되었는지 확인 할 수 있는 단계가 되겠습니다.

❑ 이러한 시뮬레이션 과정은 설계 소프트웨어 내부에서 지원하는 것을 사용해서 분석하는 방법과 각 회사의 홈페이지에서 지원하는 ModelSim의 다른 tool을 연동하여 사용하는 방법이 있습니다. 다음의 [그림 6-8]에서는 시뮬레이션을 구동하고 결과를 적어 보기 바랍니다.



[그림 6-8] 시뮬레이션 결과 표기

3) 보드에서의 확인

❑ 사용하는 디바이스는 프로젝트 선언 단계에서 미리 정해 주었기 때문에 핀에 대한 정의만 여기에서 해 주면 됩니다. 이 때 각 설계 소프트웨어를 통한 핀 맵 정의를 다음의 [표 6-1]에서 참고하여 핀을 할당해 주면 됩니다. 아래 표는 보드에서 디바이스와 각종 소자(LED와 KEY)를 연결한 것입니다.

〈표 6-1〉 핀 설정

입력				출력			
핀이름	연결소자	Altera	Xilinx	핀이름	연결소자	Altera	Xilinx
A	SW_1	Y10	AD10	Y	LED1	AF7	AB7
B	SW_2	W10	AC10				
C	SW_3	AA9	AA9				

❑ 위의 표를 참고하여 핀 설정이 끝나면 Altera Quartus II에서는 컴파일을 다시 하여 핀 설정을 프로젝트에 적용해 주고, Xilinx ISE에서는 Implement Design 하위를 컴파일 하여 핀 설정을 디바이스에 적용해 줍니다. 사전에 핀 설정에 대한 변경 사항을 미리 저장하고 이 작업을 하기 바랍니다.

❑ 핀 할당과 컴파일 과정이 끝나면 각 설계 소프트웨어의 디바이스 다운로드 창을 이용하여 디바이스에 프로그램 하면 됩니다. 따라서 Quartus II의 최종 프로그램 파일인 .sof 파일을 가지고 프로그램 하면 되고, ISE에서는 .bit 파일을 가지고 디바이스에 프로그램 하면 됩니다.

❑ 이렇게 프로그램을 한 다음에 [표 6-2]를 통해 출력 값을 적어보기 바랍니다.

〈표 6-2. 결과 확인

입 력			출 력
A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

❑ 모든 측정이 끝나면 보드의 전원을 끕니다.

4) VHDL으로의 설계

이 부분에서는 위에서 Graphic Editor를 이용하여 설계한 논리 회로를 VHDL이라는 언어를 이용해서 설계하도록 합니다.

❑ 이 방법도 앞에서 설계하는 것과 같이 Project를 선언을 먼저 합니다. 참고로 여기에서는 EX_1_1_V로 하고, 나머지 설정 부분은 Graphic Design에 한 부분과 같게 Project를 선언하면 됩니다.

❑ 메뉴의 New를 통해 VHDL로 설계하는 창을 열게 됩니다.

❑ Text Editor 창에서 아래와 같이 설계 합니다.

```
ENTITY EX_1_1_V IS
        PORT(
                A       : IN BIT;
                B       : IN BIT;
                C       : IN BIT;

                Y       : OUT BIT);
END EX_1_1_V;

ARCHITECTURE HB OF EX_1_1_V IS

        SIGNAL T_AB          : BIT;
        SIGNAL T_ABC         : BIT;

BEGIN
        T_ABC <= ( NOT C ) NOR ( A NAND B );
        Y <= NOT T_ABC;
END HB;
```

❑ 설계한 논리 회로를 저장 합니다. 저장할 파일 명은 프로젝트 명과 동일하며 확장자는 VHDL 파일을 나타내는 .VHD 입니다. 즉 EX_1_1_V.VHD로 저장하면 됩니다.

❑ 프로젝트 명이 EX_1_1_V이기 때문에 초기값으로 EX_1_1_V.VHD의 이름을 가지며, 확장자가 .VHD인 것에 주의합니다.

※ 주의 : 저장하는 파일명과 ENTITY 문에서 선언한 이름이 같지 않을 경우 컴파일 과정에서 에러가 발생됩니다.

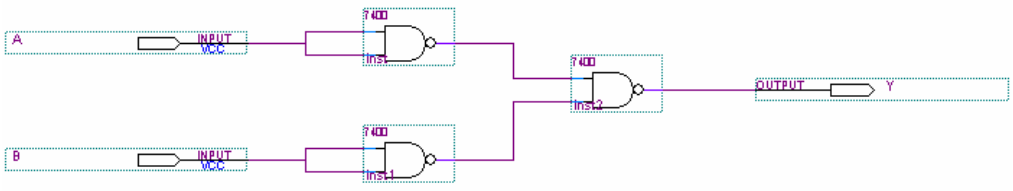
❑ 이 후의 컴파일 하는 과정 및 시뮬레이션 하는 과정은 Graphic Editor를 이용하여 설계하는 방법과 동일합니다. 위의 과정을 참고하여 컴파일에서 디바이스의 다운로드까지의 최종 과정을 다시 복습해 보기 바랍니다.

6.2.2 실습 2 : 게이트의 변환

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_1_2로 해서 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, NAND2 심볼 라이브러리를 사용하여 아래의 [그림 6-9]와 같이 논리 회로를 설계하고, EX_1_2라는 파일명으로 저장합니다.



[그림 6-9] 논리 회로

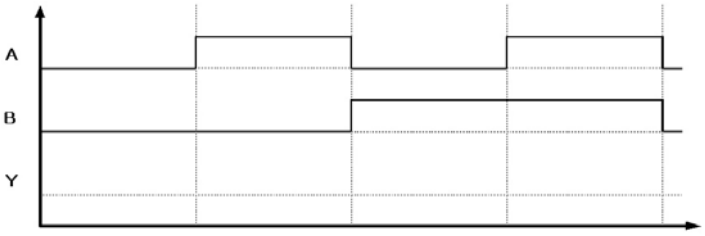
※ 참고 : VHDL로 설계한 논리 회로 EX_1_2_V.VHD 파일.

❑ 위의 회로는 아래와 같이 VHDL로 설계할 수 있으며, 저장은 EX_1_2_V.VHD 파일로 합니다.

```
ENTITY EX_1_2_V IS
    PORT(
        A, B        : IN BIT;
        Y           : OUT BIT);
END EX_1_2_V;

ARCHITECTURE HB OF EX_1_2_V IS
BEGIN

Y <= (NOT A) NAND ( NOT B);

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고, 그 결과를 [그림 6-10]에 기록합니다.



[그림 6-10] 시뮬레이션 결과 확인

2) 보드상에서의 확인

❑ [표 6-3]와 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-3〉 핀 설정

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀이름	연결소자	Altera	Xilinx
A	SW_1	Y10	AD10	Y	LED0	AF7	AB7
B	SW_2	W10	AC10				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈과 연결합니다.

❑ 보드에 전원 커넥터를 연결하고 전원을 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-4]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

※ 주의 : 보드에 장착된 스위치의 특성을 고려한다. (ON -〉 '1', OFF -〉 '0')

〈표 6-4〉 결과 확인

입 력		출 력
A	B	Y
0	0	
0	1	
1	0	
1	1	

❑ 모든 과정이 끝나면 전원을 OFF 합니다.

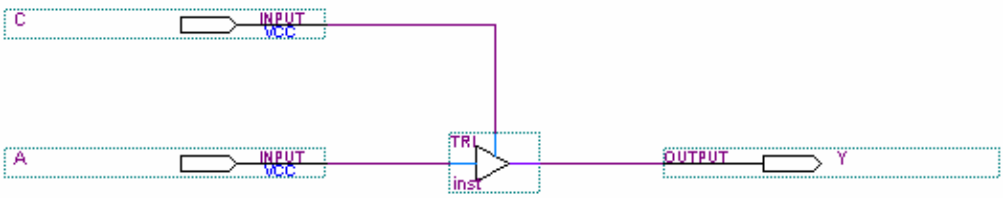
❑ 위의 표의 결과 어떤 게이트라 할 수 있습니까? : ( ) 게이트

6.2.3 실습 3 : 상태 버퍼

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard를 통해 Name을 EX_1_3로 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-11]과 같이 논리 회로를 설계하고, EX_1_3라는 파일명으로 저장합니다.



[그림 6-11] 상태 버퍼 회로도

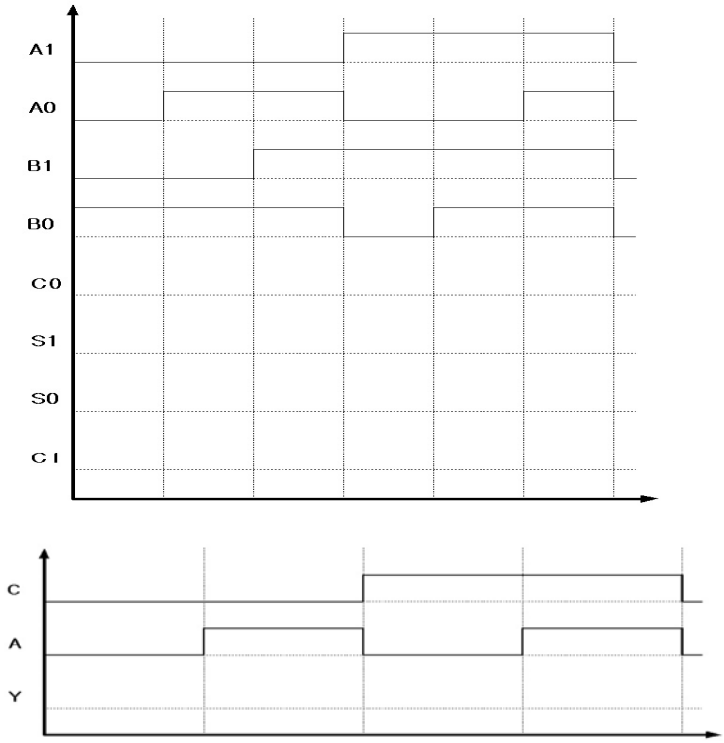
※ 참고 : VHDL로 설계 EX_1_3_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY EX_1_3_V IS
    PORT(
        A, C        : IN STD_LOGIC;
        Y           : OUT STD_LOGIC);
END EX_1_3_V;

ARCHITECTURE HB OF EX_1_3_V IS
BEGIN

Y <= A WHEN C = '1' ELSE 'Z';

END HB;
```

※ 참고 : 위의 선언에서 STD_LOGIC 이라는 TYPE이 사용되었는데, BIT TYPE이 '0'과 '1'의 값만을 선언할 수 있는 반면에, 이 STD_LOGIC TYPE은 '0', '1', 'X'(Unknown), 'Z'(High-Impedance)등의 값을 선언할 수 있습니다. 위에서 Tri-State의 경우 이 High-Impedance를 사용하기 때문에 STD_LOGIC으로 선언되었습니다.

❑ Compile하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-12]에 기록합니다.



[그림 6-12] 결과 확인

2) 보드상에서의 확인

❑ [표 6-5]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-5〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A	SW_1	Y10	AD10	Y	LED1	AF7	AB7
C	SW_2	W10	AC10				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드에 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 1-6]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

※ 주의 : 보드상에서의 스위치 특성에 주의한다. (버튼 스위치 눌렀을 때 -〉 '1', 버튼 스위치 누르지 않았을 때 -〉 '0')

입 력		출 력
A	C	Y
0	0	
0	1	
1	0	
1	1	

[표 1-6] 결과 확인

❑ 모든 과정이 끝나면 전원을 끄고, 상태 버퍼의 진리표를 참고로 결과표의 입/출력 상태를 비교해 봅니다.

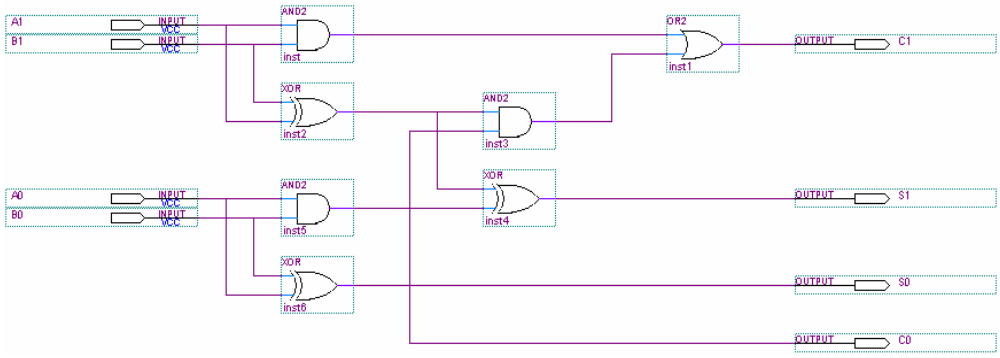
6.3 가산기와 감산기

6.3.1 실습 1 : 반가산기와 전가산기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard를 통해 Name을 EX_2_1로 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 2비트 병렬 가산기 회로를 반가산기와 전가산기를 이용하여 [그림 6-13]과 같이 설계하고, EX_2_1 이라는 파일명으로 저장합니다.



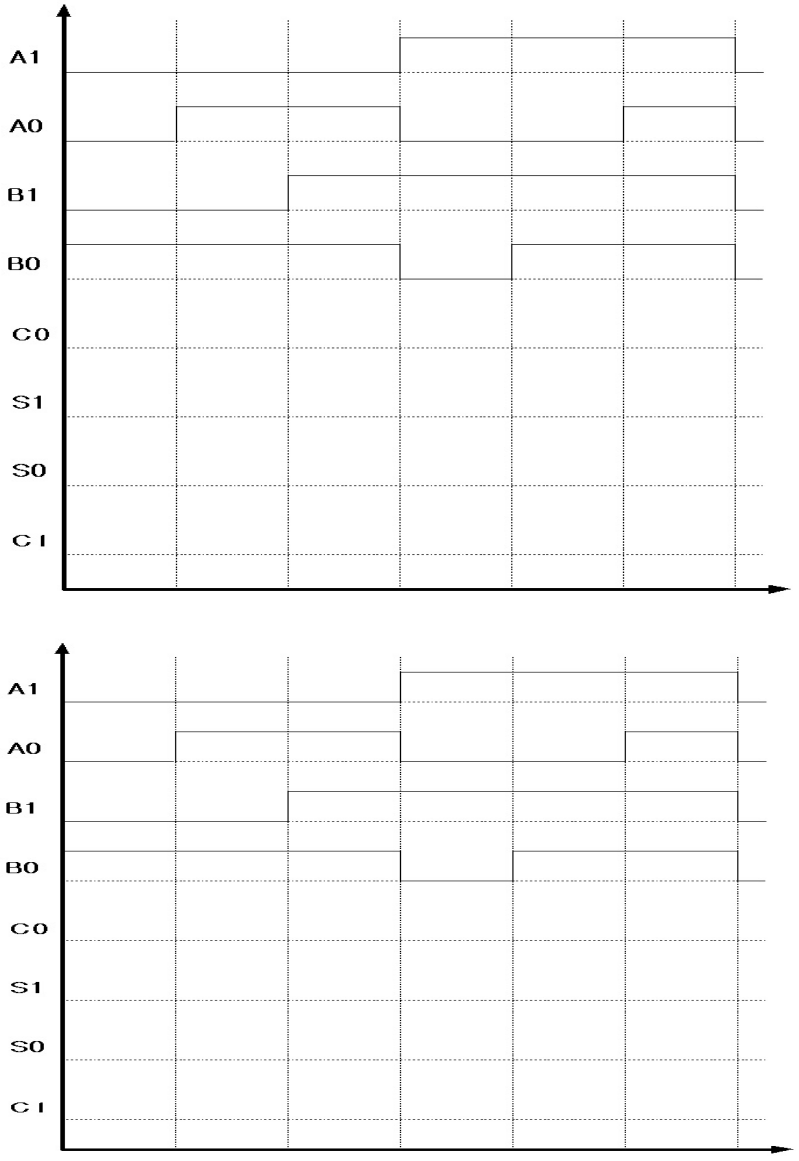
[그림 6-13] 반가산기 및 전가산기 회로도

※ 참고 : VHDL로 설계 EX_2_1_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY EX_2_1_V IS
    PORT(
        A       : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        B       : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        C       : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        S       : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END EX_2_1_V;

ARCHITECTURE HB OF EX_2_1_V IS
    SIGNAL TMP    : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN

    TMP <= A + B;
    C(1) <= TMP(2);
    C(0) <= A(0) AND B(0);
    S <= TMP(1 DOWNTO 0);

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-14]에 기록합니다.



[그림 6-14] 시뮬레이션 결과 확인

2) 보드상에서의 확인

❑ [표 6-6]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-6〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A1	SW_1	Y10	AD10	C1	LED1	AF7	AB7
A0	SW_2	W10	AC10	S1	LED2	AE7	AA7
B1	SW_3	AA9	AA9	S0	LED3	AB8	AF7
B0	SW_4	V9	Y9	C0	LED4	W8	AC7

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈과 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-7]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-7〉 결과 확인

입 력				출 력			
A1	A0	B1	B0	C1	S1	S0	C0
0	0	0	1				
0	1	0	1				
0	1	1	1				
1	0	1	0				
1	0	1	1				
1	1	1	1				

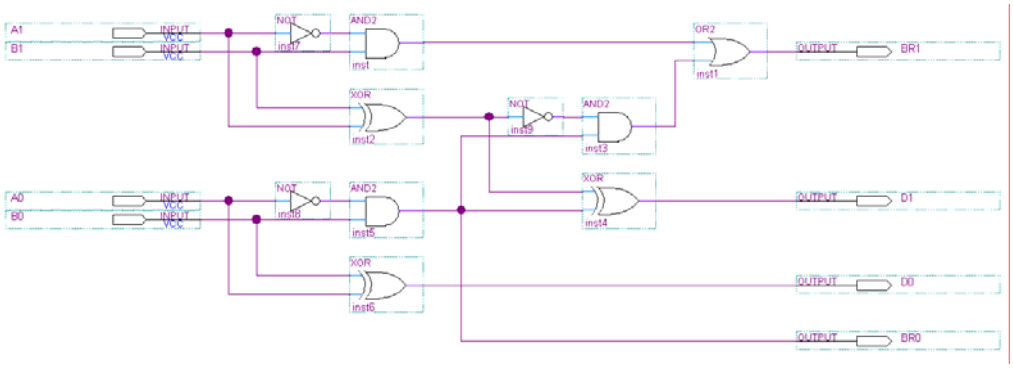
❑ 모든 과정이 끝나면 전원을 끄고, 관련 지식에 나타낸 반가산기 및 전가산기의 진리표를 참고로 결과표의 입/출력 상태를 비교해 봅니다.

6.3.2 실습 2 : 반감산기와 전감산기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_2_2로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 2비트 병렬 감산기 회로를 반감산기와 전감산기로 [그림 6-15]와 같이 설계하고, EX_2_2라는 파일명으로 저장합니다.



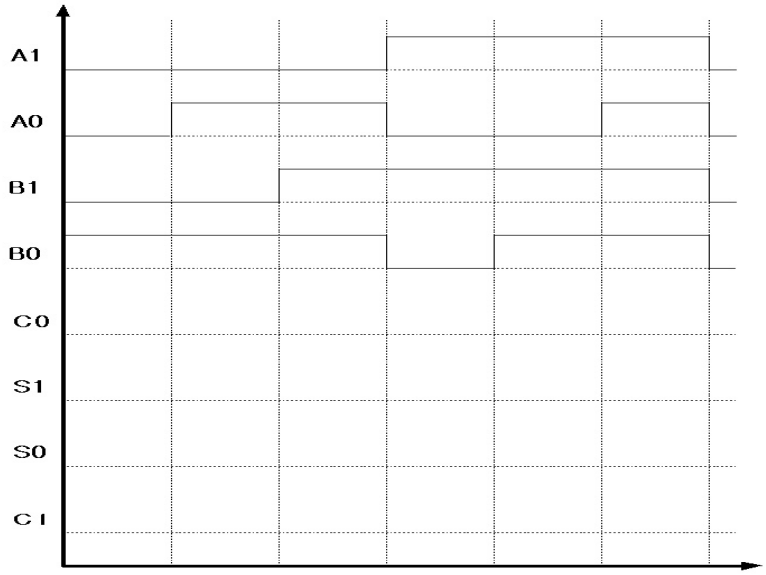
[그림 6-15] 반감산기 및 전감산기 회로도

※ 참고 : VHDL로 설계 EX_2_2_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY EX_2_2_V IS
    PORT(
        A       : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        B       : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        BR      : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        D       : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END EX_2_2_V;

ARCHITECTURE HB OF EX_2_2_V IS
    SIGNAL TMP    : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN

    TMP <= A - B;
    BR(1) <= TMP(2);
    BR(0) <= ( NOT A(0) ) AND B(0);
    D <= TMP(1 DOWNTO 0);

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-16]에 기록합니다.



[그림 6-16] 결과 확인

2) 보드상에서의 확인

❑ [표 6-8]와 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-8〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A1	SW_1	Y10	AD10	BR1	LED1	AF7	AB7
A0	SW_2	W10	AC10	D1	LED2	AE7	AA7
B1	SW_3	AA9	AA9	D0	LED3	AB8	AF7
B0	SW_4	V9	Y9	BR0	LED4	W8	AC7

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈과 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-9]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-9〉 결과 확인

입 력				출 력			
A1	A0	B1	B0	BR1	D1	D0	BR0
0	0	0	1				
0	1	0	1				
0	1	1	1				
1	0	1	0				
1	0	1	1				
1	1	1	1				

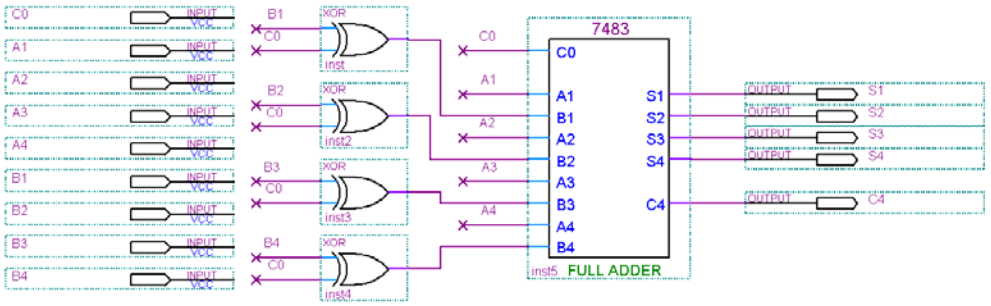
❑ 모든 과정이 끝나면 전원을 끄고, 관련 지식에 나타낸 반감산기 및 전감산기의 진리표를 참고로 결과표의 입/출력 상태를 비교해 봅니다.

6.3.3 실습 3 : 4비트 병렬 가감산기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_2_3으로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 라이브러리를 이용하여 4비트 병렬 가감산기 회로를 [그림 6-17]과 같이 설계하고, EX_2_3라는 파일명으로 저장합니다. (Xilinx symbol name : acc4)



[그림 6-17] 4비트 병렬 가감산기 회로도

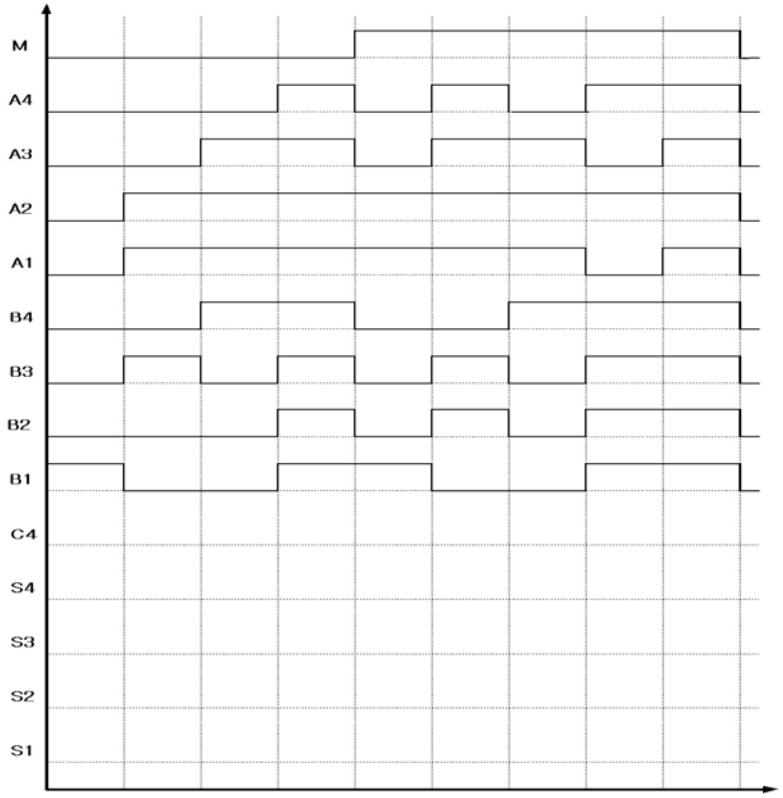
※ 주의 : 회로를 간단하게 하기 위하여 Name matching(노드명으로 연결) 방법을 사용하였습니다.

※ 참고 : VHDL로 설계 EX_2_3_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY EX_2_3_V IS
    PORT(
        C0      : IN STD_LOGIC;
        A       : IN STD_LOGIC_VECTOR(4 DOWNTO 1);
        B       : IN STD_LOGIC_VECTOR(4 DOWNTO 1);
        S       : OUT STD_LOGIC_VECTOR(4 DOWNTO 1);
        C4      : OUT STD_LOGIC);
END EX_2_3_V;

ARCHITECTURE HB OF EX_2_3_V IS
    SIGNAL TMP    : STD_LOGIC_VECTOR(5 DOWNTO 1);
BEGIN

PROCESS(C0, A, B)
BEGIN
    IF C0 = '0' THEN
        TMP <= A + B;
```

```
    ELSIF C0 = '1' THEN
        TMP <= A – B;
    END IF;
END PROCESS;

C4 <= TMP(5);
S <= TMP(4 DOWNTO 1);

END HB;
```

- Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.
- 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-18]에 기록합니다.



[그림 6-18] 결과 확인

2) 보드상에서의 확인

- [표 6-10]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-10〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
C0	SW8	W12	AB14	S1	LED1	AF7	AB7
A1	SW0	Y13	Y16	S2	LED2	AE7	AA7
A2	SW1	AB12	AB15	S2	LED3	AB8	AF7
A3	SW2	AA12	AA15	S4	LED4	W8	AC7
A4	SW3	AD12	AE15	C4	LED6	AF6	AC6
B1	SW4	AC12	AD15				
B2	SW5	U12	AF13				
B3	SW6	AE11	AA13				
B4	SW7	Y12	W15				

- ❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ Switch의 상태를 [표 6-11]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-11〉 결과 확인

SW	데이터 A					데이터 B					출 력						연산식
C_0	A_4	A_3	A_2	A_1	10진	B_4	B_3	B_2	B_1	10진	C_4	S_4	S_3	S_2	S_1	10진	10진수로 작성
0	0	0	0	0	0	0	0	0	1	1							
0	0	0	1	1	3	0	1	0	0	4							
0	0	1	1	1	7	1	0	0	0	8							
0	1	1	1	1	15	1	1	1	1	15							
1	0	0	1	1	3	0	0	0	1	1							
1	1	1	1	1	15	0	1	1	0	6							
1	0	1	1	1	7	1	0	0	0	8							
1	1	0	1	0	10	1	1	1	1	15							
1	1	1	1	1	15	1	1	1	1	15							

- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표에서 입력에 따른 결과를 살펴보고 연산 식을 10진수를

사용하여 표에 나타냅니다.

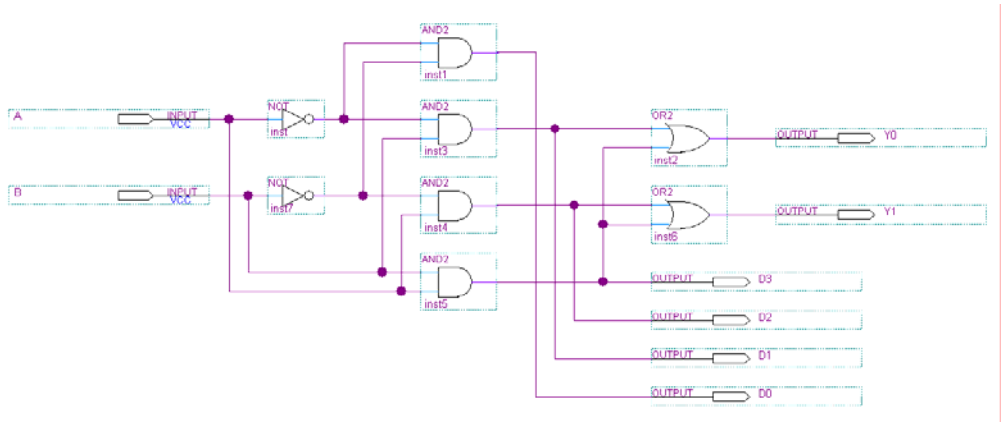
6.4 해독기와 부호기

6.4.1 실습 1 : 논리 게이트로 구성한 해독기와 부호기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_3_1로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 라이브러리를 이용하여 2 X 4 해독기와 4 X 2 부호기 회로를 [그림 6-19]과 같이 설계하고, EX_3_1라는 파일명으로 저장합니다.



[그림 6-19] 해독기 및 부호기 회로도

※ 참고 : VHDL 파일로 설계 EX_3_1_V.VHD 파일

```
ENTITY EX_3_1_V IS
    PORT(
        A, B   : IN   BIT;
        D    : OUT BIT_VECTOR(3 DOWNTO 0);
        Y : OUT BIT_VECTOR(1 DOWNTO 0));
END EX_3_1_V;

ARCHITECTURE HB OF EX_3_1_V IS
BEGIN

PROCESS(A,B)
BEGIN
    IF A = '0' AND B = '0' THEN
        D <= "0001";
    ELSIF A = '0' AND B = '1' THEN
        D <= "0010";
    ELSIF A = '1' AND B = '0' THEN
        D <= "0100";
    ELSIF A = '1' AND B = '1' THEN
```

```
ENTITY EX_3_1_V IS
    PORT(
        A, B     : IN   BIT;
        D          : OUT  BIT_VECTOR(3  DOWNTO
0);
        Y          : OUT  BIT_VECTOR(1  DOWNTO
0));
END EX_3_1_V;

ARCHITECTURE HB OF EX_3_1_V IS
    SIGNAL TMP : BIT_VECTOR(1 DOWNTO 0);
BEGIN

TMP <= A & B;

PROCESS(TMP)
BEGIN
    CASE TMP IS
        WHEN "00" => D <= "0001";
```

```
        D <= "1000";
    END IF;
END PROCESS;

Y <= A & B;

END HB;
```

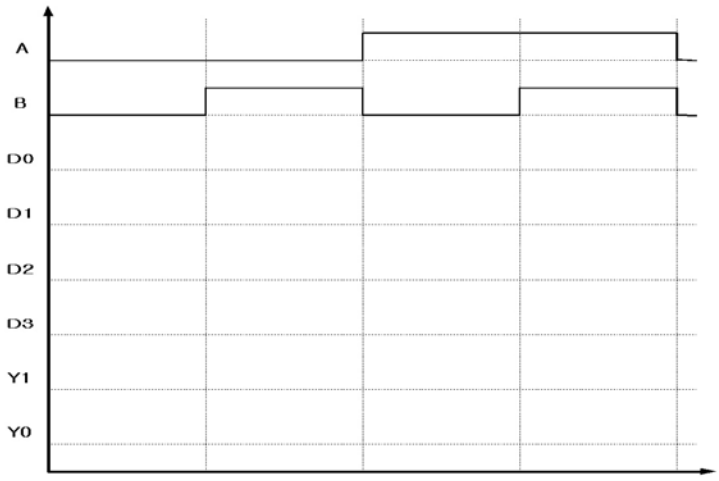
```
        WHEN "01" => D <= "0010";
        WHEN "10" => D <= "0100";
        WHEN "11" => D <= "1000";
    END CASE;
END PROCESS;

Y <= A & B;

END HB;
```

※ 참고 : 위에서 VHDL로 설계된 두 논리 회로의 결과는 같습니다. 차이는 IF문을 사용한 것과 CASE문을 사용한 것의 차이인데, 이 차이 IF문은 포괄적인 제어 분석을 하는 경우에 사용되고, CASE문은 각각에 대해서 조건 제어를 하는 것이 특징입니다. 이 논리 회로의 결과에서는 크게 차이가 나타나지 않지만, 컴파일 과정에서 합성을 하게 되면, IF문은 간단한 Control Logic으로 변환되고 CASE문은 MUX형태로 변하게 되는 차이가 있습니다.

※ 참고 : 위의 BIT_VECTOR 선언에서 "D : OUT BIT_VECTOR(3 DOWNTO 0);" 부분에서 (3 DOWNTO 0)의 앞(3)에는 최상위 비트가 될 부분을 씁니다. 예를 들어 A(3 DOWNTO 0)과 A(0 TO 3)의 차이는 각각의 상위 비트가 A(3)인지 A(0)인지의 차이입니다.

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-20]에 기록합니다.



[그림 6-20] 결과 확인

2) 보드상에서의 확인

❑ [표 6-12]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-12〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A	SW_1	Y10	AD10	D0	LED1	AF7	AB7
B	SW_2	W10	AC10	D1	LED2	AE7	AA7
				D2	LED3	AB8	AF7
				D3	LED4	W8	AC7
				Y0	LED5	AF6	AD6
				Y1	LED6	AE6	AC6

- ❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈과 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ Switch의 상태를 [표 6-13]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-13〉 결과 확인

입 력		해독기 출력				부호기 출력	
A	B	D0	D1	D2	D3	Y1	Y0
0	0						
0	1						
1	0						
1	1						

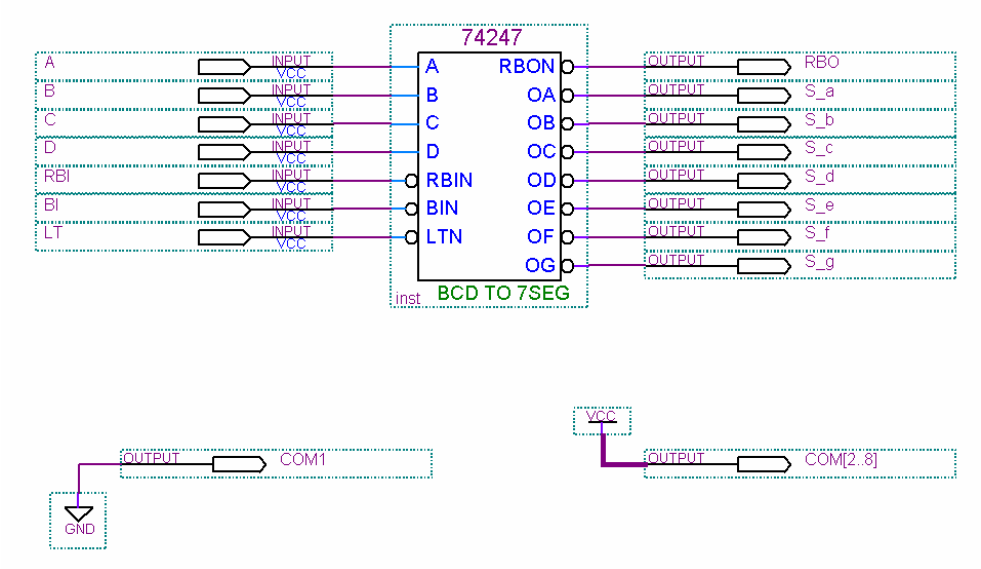
- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

6.4.2 실습 2 : BCD-7 세그먼트 해독기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_3_2로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-21]과 같이 설계하고, EX_3_2라는 파일명으로 저장합니다.



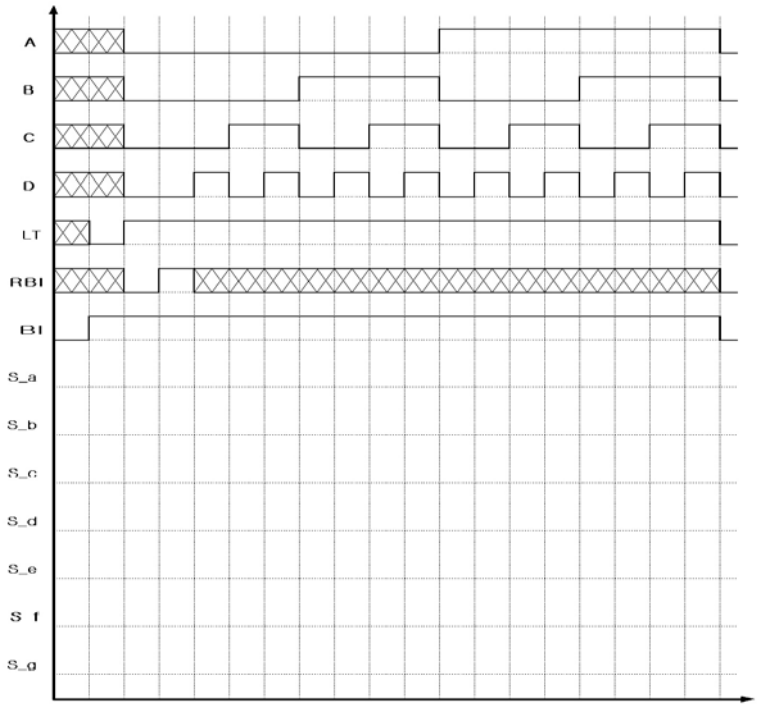
[그림 6-21] BCD-7 세그먼트 해독기

※ 참고 : VHDL 파일로 설계 EX_3_2_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY EX_3_2_V IS
    PORT(
        A, B, C, D              : IN STD_LOGIC;
        LT, RBI, BI             : IN STD_LOGIC;
        S_a, S_b, S_c, S_d      : OUT STD_LOGIC;
        S_e, S_f, S_g, RBO      : OUT STD_LOGIC;
com                     : OUT STD_LOGIC_VECTOR(1 to 6));
END EX_3_2_V;

ARCHITECTURE HB OF EX_3_2_V IS

    SIGNAL TMP_D    : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL TMP      : STD_LOGIC_VECTOR(6 DOWNTO 0);
BEGIN
```

```
    TMP_D <= A & B & C & D;
    com1 <= '0';
    com(2 to 6) <= "11111";

PROCESS(LT, RBI, BI, TMP)
BEGIN
    IF LT = '0' AND BI = '1' THEN
        TMP <= "1000000";
        RBO <= '1';
    ELSIF LT = '1' AND RBI = '0' AND TMP_D = "0000" THEN
        TMP <= "1111111";
        RBO <= '0';
    ELSIF BI = '0' THEN
        TMP <= "1111111";
    ELSIF LT = '1' AND BI = '1' THEN
        CASE TMP_D IS
            WHEN "0000" =>   TMP <= "1111110";
                             RBO <= '1';
            WHEN "0001" =>   TMP <= "0110000";
                             RBO <= '1';
            WHEN "0010" =>   TMP <= "1101101";
                             RBO <= '1';
            WHEN "0011" =>   TMP <= "1111001";
                             RBO <= '1';
            WHEN "0100" =>   TMP <= "0110011";
                             RBO <= '1';
            WHEN "0101" =>   TMP <= "1011011";
                             RBO <= '1';
            WHEN "0110" =>   TMP <= "1011111";
                             RBO <= '1';
            WHEN "0111" =>   TMP <= "1110000";
                             RBO <= '1';
            WHEN "1000" =>   TMP <= "1111111";
                             RBO <= '1';
            WHEN "1001" =>   TMP <= "1111011";
                             RBO <= '1';
            WHEN OTHERS =>
        END CASE;
    END IF;
END PROCESS;

S_a <= TMP(6);
S_b <= TMP(5);
S_c <= TMP(4);
S_d <= TMP(3);
S_e <= TMP(2);
S_f <= TMP(1);
S_g <= TMP(0);

END HB;
```

- Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.
- 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-22]에 기록합니다.



[그림 6-22] 결과 확인

2) 보드상에서의 확인

❑ [표 6-14]과 같이 각 입력 / 출력 포트의 핀 번호를 설정한 후 컴파일 한다.

〈표 6-14〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A	SW_1	Y10	AD10	S_a	A	AF5	AF5
B	SW_2	W10	AC10	S_b	B	AE5	AE5
C	SW_3	AA9	AA9	S_c	C	AD6	AB6
D	SW_4	V9	Y9	S_d	D	AC6	AA6
LT	SW0	Y13	Y16	S_e	E	AA2	K26
RBI	SW1	AB12	AB15	S_f	F	AA1	K25
BI	SW2	AA12	AA15	S_g	G	AA6	AC2
				com1	COM1	Y1	AD2
				com2	COM2	Y4	W2
				com3	COM3	Y3	W1
				com4	COM4	W1	AB4
				com5	COM5	Y5	AB3
				com6	COM6	W3	W6
				RBO	LED1	AF7	AB7

- ❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈과 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ Switch의 상태를 [표 6-15]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-15〉 결과 확인

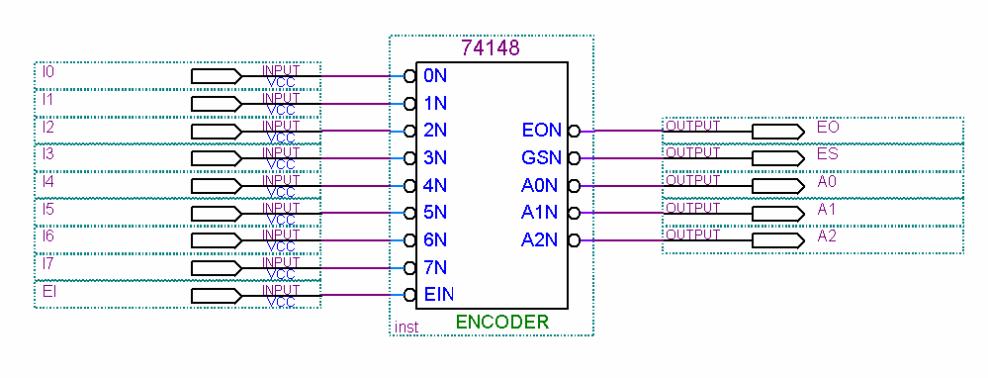
10진수	BCD 입력				제어 입력			출 력								표시기 표시
	D	C	B	A	LT	RBI	BI	S_a	S_b	S_c	S_d	S_e	S_f	S_g	RBO	
	X	X	X	X	X	X	0									
	X	X	X	X	0	X	1									
0	0	0	0	0	1	0	1									
0	0	0	0	0	1	1	1									
1	0	0	0	1	1	X	1									
2	0	0	1	0	1	X	1									
3	0	0	1	1	1	X	1									
4	0	1	0	0	1	X	1									
5	0	1	0	1	1	X	1									
6	0	1	1	0	1	X	1									
7	0	1	1	1	1	X	1									
8	1	0	0	0	1	X	1									
9	1	0	0	1	1	X	1									
10	1	0	1	0	1	X	1									
11	1	0	1	1	1	X	1									
12	1	1	0	0	1	X	1									
13	1	1	0	1	1	X	1									
14	1	1	1	0	1	X	1									
15	1	1	1	1	1	X	1									

- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

6.4.3 실습 3 : 8 X 3 부호기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

- ❑ New Project Wizard에서 Name을 EX_3_3로 하여 프로젝트를 생성합니다.
- ❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-23]과 같이 설계하고, EX_3_3라는 파일명으로 저장합니다.



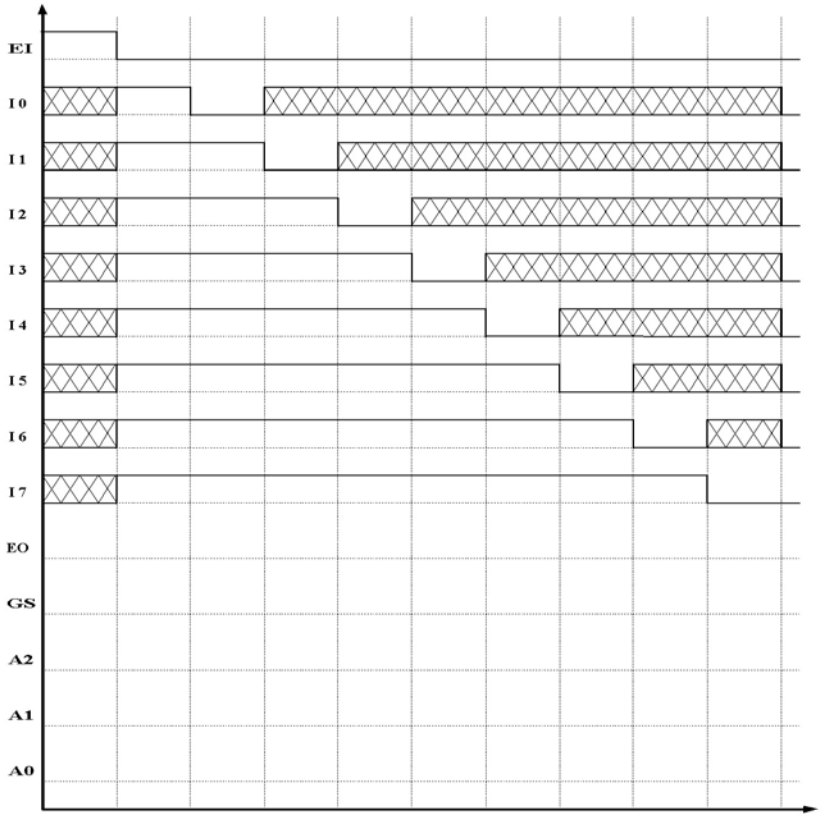
[그림 6-23] 8x3 부호기 회로도

※ 참고 : VHDL 파일로 설계 EX_3_3_V.VHD 파일

```
ENTITY EX_3_3_V IS
    PORT(
        I           : IN BIT_VECTOR(7 DOWNTO 0);
        EI          : IN BIT;
        EO, GS      : OUT BIT;
        A           : OUT BIT_VECTOR(2 DOWNTO 0));
END EX_3_3_V;
ARCHITECTURE HB OF EX_3_3_V IS
BEGIN

PROCESS(I,EI)
BEGIN
    IF EI = '1' THEN
        A <= "111";
        GS <= '1';
        EO <= '1';
    ELSIF I ="11111111" THEN
        A <= "111";
        GS <= '1';
        EO <= '0';
    ELSIF I(7) = '0' THEN
            A <= "000";
        GS <= '0';
        EO <= '1';
    ELSIF I(7 DOWNTO 6) = "10" THEN
            A <= "001";
```

```
            GS <= '0';
            EO <= '1';
        ELSIF I(7 DOWNTO 5) = "110" THEN
                A <= "010";
            GS <= '0';
            EO <= '1';
        ELSIF I(7 DOWNTO 4) = "1110" THEN
                A <= "011";
            GS <= '0';
            EO <= '1';
        ELSIF I(7 DOWNTO 3) = "11110" THEN
                A <= "100";
            GS <= '0';
            EO <= '1';
        ELSIF I(7 DOWNTO 2) = "111110" THEN
                A <= "101";
            GS <= '0';
            EO <= '1';
        ELSIF I(7 DOWNTO 1) = "1111110" THEN
                A <= "110";
            GS <= '0';
            EO <= '1';
        ELSIF I(7 DOWNTO 0) = "11111110" THEN
                A <= "111";
            GS <= '0';
            EO <= '1';
        END IF;
    END PROCESS;

    END HB;
```

- Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.
- 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-24]에 기록합니다.



[그림 6-24] 결과 확인

2) 보드상에서의 확인

❑ [표 6-16]와 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-16〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
I0	SW0	Y13	Y16	EO	LED1	AF7	AB7
I1	SW1	AB12	AB15	GS	LED2	AE7	AA7
I2	SW2	AA12	AA15	A0	LED3	AB8	AF7
I3	SW3	AD12	AE15	A1	LED4	W8	AC7
I4	SW4	AC12	AD15	A2	LED5	AF6	AD6
I5	SW5	U12	AF13				
I6	SW6	AE11	AA13				
I7	SW7	Y12	W15				
EI	SW8	W12	AB14				

- ❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ Switch의 상태를 [표 6-17]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-17〉 결과 확인

입 력									출 력				
EI	I0	I1	I2	I3	I4	I5	I6	I7	EO	GS	A2	A1	A0
1	X	X	X	X	X	X	X	X					
0	1	1	1	1	1	1	1	1					
0	0	1	1	1	1	1	1	1					
0	X	0	1	1	1	1	1	1					
0	X	X	0	1	1	1	1	1					
0	X	X	X	0	1	1	1	1					
0	X	X	X	X	0	1	1	1					
0	X	X	X	X	X	0	1	1					
0	X	X	X	X	X	X	0	1					
0	X	X	X	X	X	X	X	0					

- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

6.5 채널 선택 및 분배 회로

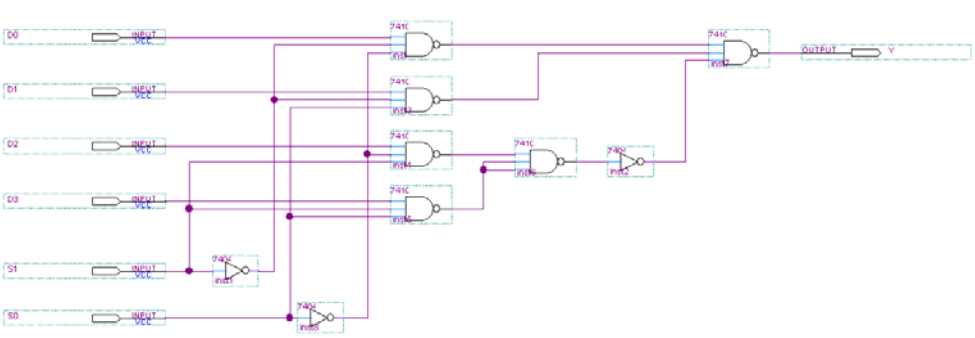
6.5.1 실습 1 : 채널 선택 회로

1) 4채널 선택회로

① Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_4_1A로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 4 채널 선택 회로를 [그림 6-25]과 같이 설계하고, EX_4_1A라는 파일명으로 저장합니다.



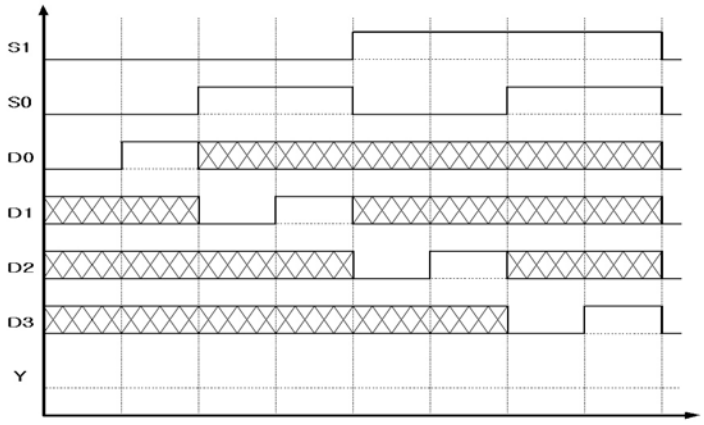
[그림 6-25] 4채널 선택회로 회로도

※ 참고 : VHDL 파일로 설계 EX_4_1A_V.VHD 파일

```
ENTITY EX_4_1A_V IS
    PORT(
        D       : IN BIT_VECTOR(0 TO 3);
        S       : IN BIT_VECTOR(0 TO 1);
        Y       : OUT BIT);
END EX_4_1A_V;

ARCHITECTURE HB OF EX_4_1A_V IS
BEGIN

PROCESS(D, S)
BEGIN
    CASE S IS
        WHEN “00” => Y <= D(0);
        WHEN “01” => Y <= D(1);
        WHEN “10” => Y <= D(2);
        WHEN “11” => Y <= D(3);
    END CASE;
END PROCESS;

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-26]에 기록합니다.



[그림 6-26] 결과 확인

② 보드상에서의 확인

❑ 아래 [표 6-18]와 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-18〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
D0	SW_1	Y10	AD10	Y	LED1	AF7	AB7
D1	SW_2	W10	AC10				
D2	SW_3	AA9	AA9				
D3	SW_4	V9	Y9				
S1	SW0	Y13	Y16				
S0	SW1	AB12	AB15				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블을 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-19]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션

의 결과와 비교합니다.

〈표 6-19〉 결과 확인

선 택 신 호		입 력				출 력
S1	S0	D0	D1	D2	D3	Y
0	0	0	X	X	X	
0	0	1	X	X	X	
0	1	X	0	X	X	
0	1	X	1	X	X	
1	0	X	X	0	X	
1	0	X	X	1	X	
1	1	X	X	X	0	
1	1	X	X	X	1	

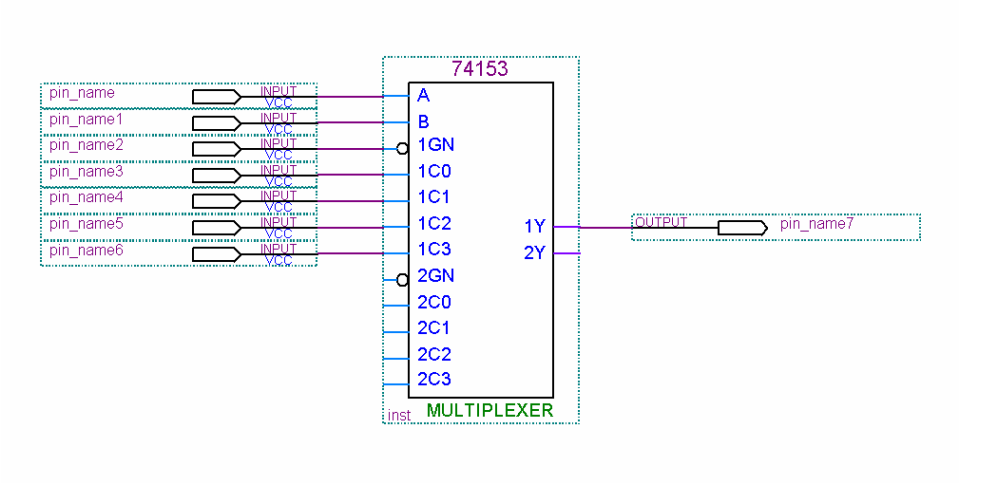
❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

2) 4 X 1 멀티 플렉서

① Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard를 통해 Name을 EX_4_1B로 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-27]과 같이 설계하고, EX_4_1B라는 파일명으로 저장합니다.

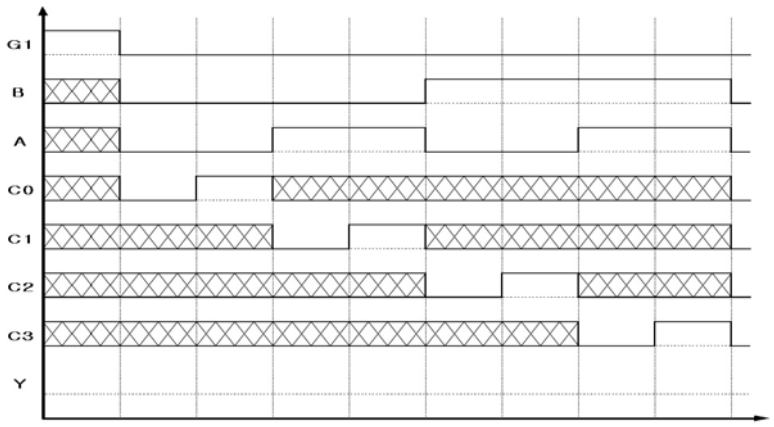


[그림 6-27] 4x1 멀티플렉서 회로도

※ VHDL 파일은 **A. 4채널 선택회로**에서 설계한 파일과 동일합니다.

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-28]에 기록합니다.



[그림 6-28] 결과 확인

② **보드상에서의 확인**

❑ [표 6-20]와 같이 각 입력 / 출력 포트의 핀 번호를 설정한 후 컴파일 합니다.

〈표 6-20〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
G1	SW_1	Y10	AD10	Y	LED1	AF7	AB7
A	SW_2	W10	AC10				
B	SW_3	AA9	AA9				
C0	SW0	Y13	Y16				
C1	SW1	AB12	AB15				
C2	SW2	AA12	AA15				
C3	SW3	AD12	AE15				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-21]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-21〉 결과 확인

선택 신호			입 력				출 력
G1	B	A	C0	C1	C2	C3	Y
1	X	X	X	X	X	X	
0	0	0	0	X	X	X	
0	0	0	1	X	X	X	
0	0	1	X	0	X	X	
0	0	1	X	1	X	X	
0	1	0	X	X	0	X	
0	1	0	X	X	1	X	
0	1	1	X	X	X	0	
0	1	1	X	X	X	1	

❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

❑ EX_4_1A의 4 채널 선택 회로에서의 결과값과 EX_4_1B의 4 X 1 멀티 플렉서에서의 결과값을 비교해 봅니다.

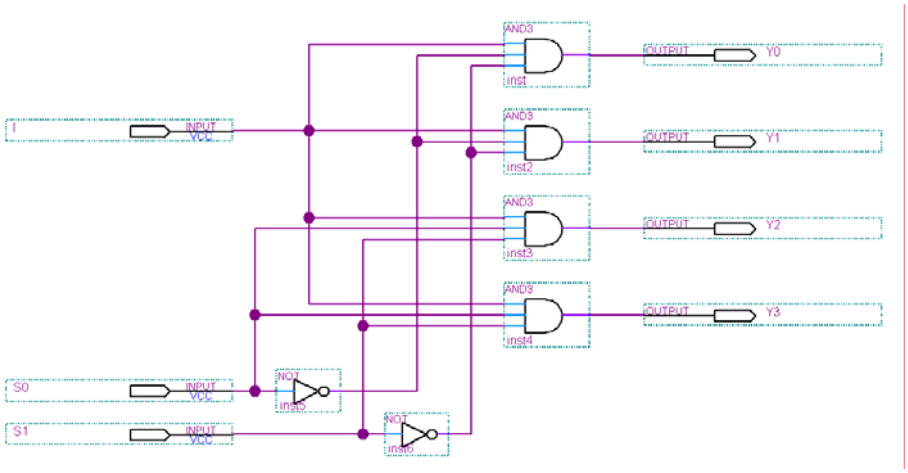
6.5.2 실습 2 : 채널 분배 회로

1) 4채널 분배회로

① Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_4_2A로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 4채널 분배 회로를 [그림 6-29]과 같이 설계하고, EX_4_2A라는 파일명으로 저장합니다.



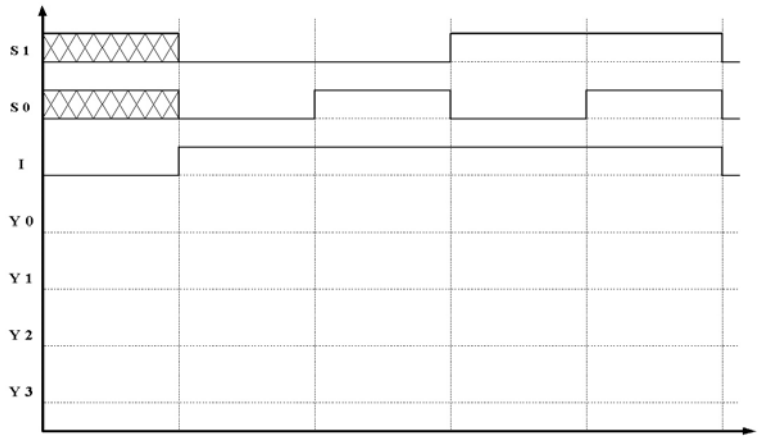
[그림 6-29] 4채널 분배회로 회로도

※ 참고 : VHDL 파일로 설계 EX_4_2A_V.VHD 파일

```
ENTITY EX_4_2B_V IS
    PORT(
        I       : IN BIT;
        S       : IN BIT_VECTOR(0 T0 1);
        Y       : OUT BIT_VECTOR(0 TO 3));
END EX_4_2B_V;

ARCHITECTURE HB OF EX_4_2B_V IS
BEGIN

PROCESS(I,S)
BEGIN
    CASE S IS
        WHEN “00” => Y(0) <= I;
        WHEN “01” => Y(1) <= I;
        WHEN “10” => Y(2) <= I;
        WHEN “11” => Y(3) <= I;
    END CASE;
END PROCESS;

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-30]에 기록합니다.



[그림 6-30] 결과 확인

② 보드상에서의 확인

❑ [표 6-22]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-22〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
I	SW0	Y13	Y16	Y0	LED1	AF7	AB7
S0	SW_1	Y10	AD10	Y1	LED2	AE7	AA7
S1	SW_2	AA9	AC10	Y2	LED3	AB8	AF7
				Y3	LED4	W8	AC7

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-23]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-23〉 결과 확인

선택 신호		입 력	출 력			
S1	S0	I	Y0	Y1	Y2	Y3
X	X	0				
0	0	1				
0	1	1				
1	0	1				
1	1	1				

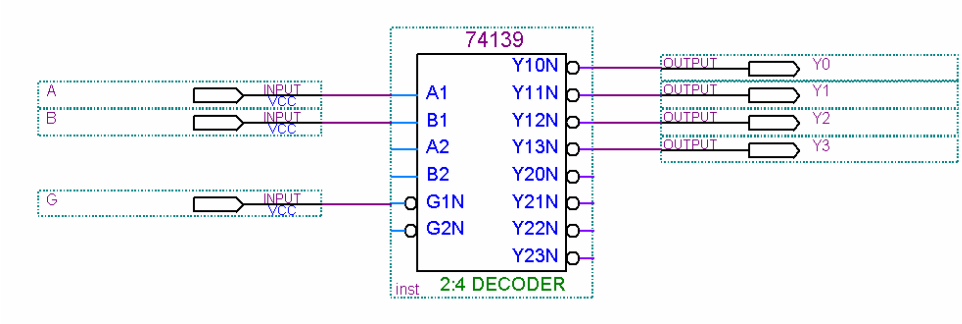
❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

2) 74139를 사용한 채널 분배회로

① Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_4_2B로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-31]과 같이 설계하고, EX_4_2B라는 파일명으로 저장합니다.

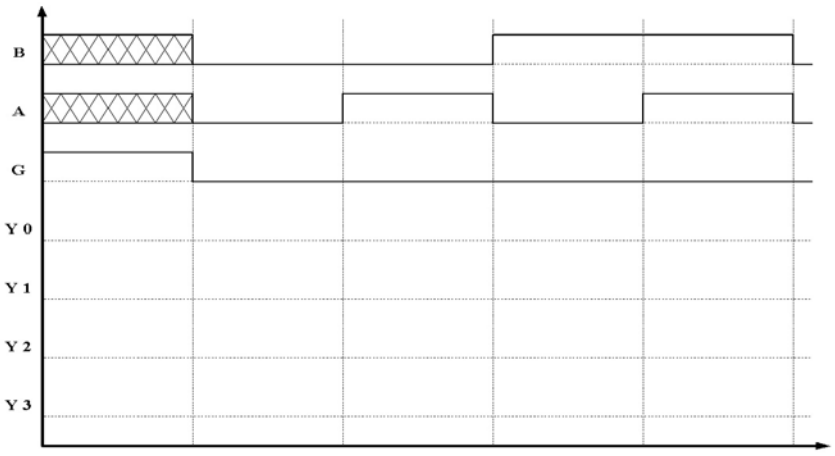


[그림 6-31] 74139를 사용한 회로도

※ VHDL 파일은 **A. 4채널 분배 회로**에서 설계한 파일과 같다.

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-32]에 기록합니다.



[그림 6-32] 결과 확인

② 보드상에서의 확인

❑ [표 6-24]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-24〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A	SW_1	Y10	AD10	Y0	LED1	AF7	AB7
B	SW_2	W10	AC10	Y1	LED2	AE7	AA7
G	SW0	Y13	Y16	Y2	LED3	AB8	AF7
				Y3	LED4	W8	AC7

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-25]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-25〉 결과 확인

선택 신호		입 력	출 력			
B	A	G	Y0	Y1	Y2	Y3
X	X	1				
0	0	0				
0	1	0				
1	0	0				
1	1	0				

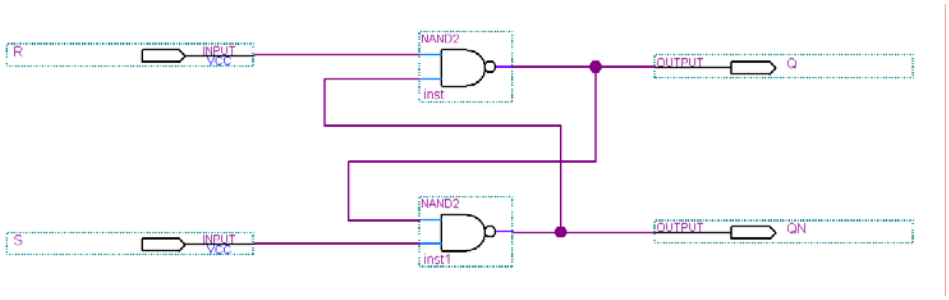
❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

❑ EX_4_2A의 4 채널 분배 회로에서의 결과값과 EX_4_2B의 74139를 사용한 분배회로에의 결과값을 비교해 봅니다.

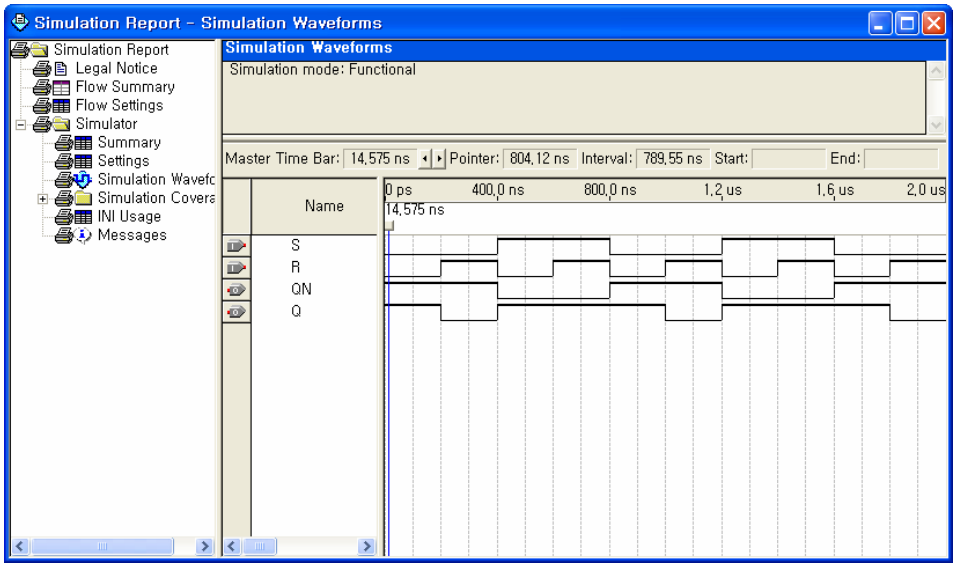
6.6 플립 플롭과 시프트 레지스터

6.6.1 실습 1 : RS F/F

RS F/F을 설계하는 방법은 NAND 게이트를 이용한 방법, NOT 게이트를 이용한 방법 등이 있는데, PLD는 특성상 어떤 값이 피드백 되어 자신의 값에 영향을 미치는 논리 회로를 설계하는데 문제가 있습니다. 예를 들어 아래와 같이 NAND 게이트를 이용하여 RS F/F를 설계하였을 때, 결과치는 원래의 RS F/F의 결과값과는 다른 엉뚱한 값이 출력되는 것을 볼 수 있습니다.



[그림 6-33] NAND 게이트를 이용한 RS F/F 설계



[그림 6-34] 위 회로에 대한 결과

위의 [그림 6-34]에서는 Altera Quartus II에서 이러한 것을 시뮬레이션 했을 때 결과를

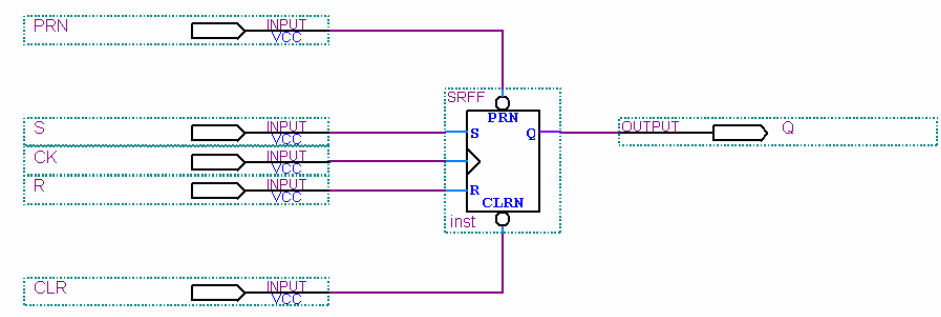
보여주고 있습니다. 여기에서는 결과가 전혀 엉뚱하게 시뮬레이션이 되는 것을 확인할 수 있습니다. 이렇기 때문에 PLD를 설계할 때에는 F/F 심볼 라이브러리를 따로 불러와 사용해야 합니다.

이 실험에서는 그래픽 에디터에서 RS F/F의 심볼 라이브러리를 불러와서 실험하고, 브래드 보드에서는 7400(NAND)를 사용하여 실험하여 그 결과를 비교하도록 하겠습니다.

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_5_1로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-35]과 같이 논리 회로를 설계하고, EX_5_1라는 파일명으로 저장합니다. 다음의 그림은 Quartus II에서 작업한 모습을 보여주고 있습니다. (참고로 심볼 명은 Quartus II 에서는 "SRFF"이고, ISE 에서는 "FDRS"를 불러와서 사용하면 됩니다.)



[그림 6-35] 회로도

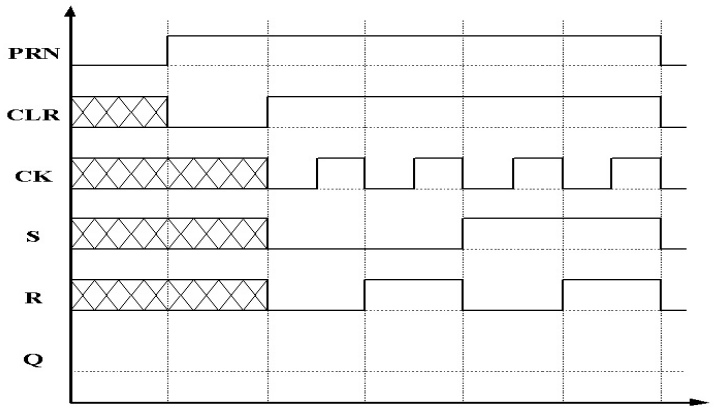
※ 참고 : VHDL 파일로 설계 EX_5_1_V.VHD 파일

```
ENTITY EX_5_1_V IS
    PORT(
        PRN, CLR        : IN BIT;
        R, S, CK        : IN BIT;
        Q               : OUT BIT );
END EX_5_1_V;
ARCHITECTURE HB OF EX_5_1_V IS
    SIGNAL TMP    : BIT;
BEGIN
PROCESS(PRN, CLR, R, S, CK)
BEGIN
    IF PRN = '0' THEN
        TMP <= '1';
    ELSIF CLR = '0' THEN
        TMP <= '0';
    ELSIF CK'EVENT AND CK = '1' THEN
        IF R = '0' AND S = '0' THEN
            TMP <= TMP;
        ELSIF R = '0' AND S = '1' THEN
            TMP <= '1';
        ELSIF R = '1' AND S = '0' THEN
            TMP <= '0';
        ELSIF R = '1' AND S = '1' THEN
            TMP <= NOT TMP;
        END IF;
    END IF;
END PROCESS;
```

```
Q <= TMP;

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-36]에 기록합니다.



[그림 6-36] 결과 확인

2) 보드상에서의 확인

❑ [표 6-26]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-26〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
PRN	SW0	Y13	AD10	Q	LED1	AF7	AB7
CLR	SW1	AB12	AC10				
S	SW_1	Y10	Y16				
R	SW_2	W10	AB15				
CK	SW_3	AA9	AA15				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-27]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-27〉 결과 확인

입 력					출 력
CLR	PR	CK	S	R	Q
0	1	X	X	X	
1	0	X	X	X	
1	1	↑	0	0	
		↑	0	1	
		↑	1	0	
		↑	1	1	

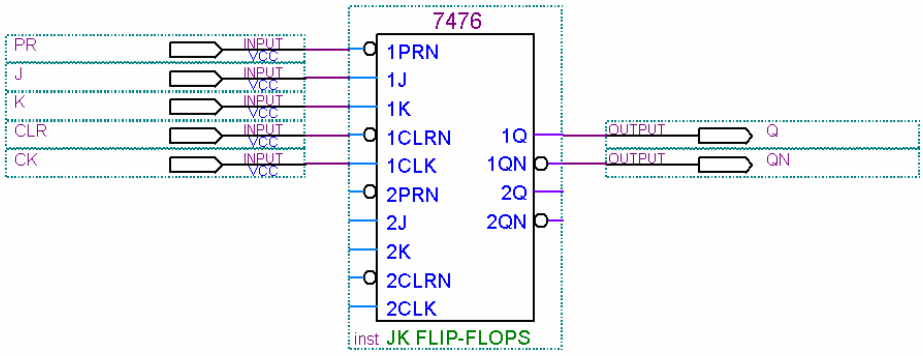
※ 주의 : [표 5-3]에서 'X'의 표시일 경우 입력 상태를 '0'이나 '1'로 바꾸어 보며 출력의 변화 유무를 확인하고, '↑'는 클록 펄스의 상승 에지 즉 CK의 데이터가 '0'에서 '1'로 변하는 순간에 F/F가 동작한다는 뜻입니다.

❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

6.6.2 실습 2 : JK F/F

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

- ❑ New Project Wizard에서 Name을 EX_5_2로 하여 프로젝트를 생성합니다.
- ❑ Block Diagram/Schematic File에서 심볼 라이브러리를 이용하여 [그림 6-37]과 같이 논리 회로를 설계하고, EX_5_2라는 파일명으로 저장합니다. 다음의 설계 파일은 Quartus II에서 설계한 모습을 보여주고 있습니다. (ISE 에서는 "FJKPE"의 심볼을 사용하여 설계합니다.)



[그림 6-37] JK F/F 회로도

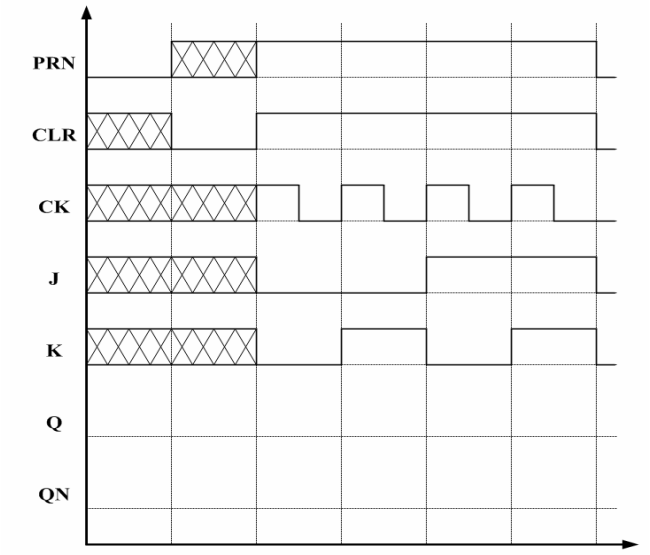
※ 참고 : VHDL 파일로 설계 EX_5_2_V.VHD 파일

```
ENTITY JKPR IS
    PORT(
        CLK, J , K, PR, CLR     : IN BIT;
        Q     : BUFFER BIT );
END JKPR;

ARCHITECTURE HB OF JKPR IS
BEGIN

PROCESS(CLK, PR, CLR)
BEGIN
    IF PR = '0' THEN
        Q <= '1';
    ELSIF CLR = '0' THEN
        Q <= '0';
    ELSIF CLK'EVENT AND CLK = '1' THEN
        IF J = '1' AND K = '0' THEN
            Q <= '1';
        ELSIF J = '0' AND K = '1' THEN
            Q <= '0';
        ELSIF J = '1' AND K = '1' THEN
            Q <= NOT Q;
        END IF;
```

```
    END IF;
END PROCESS;

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-38]에 기록합니다.



[그림 6-38] 결과 확인

2) 보드상에서의 확인

❑ [표 6-28]와 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-28〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
PRN	SW0	Y13	AD10	Q	LED1	AF7	AB7
CLR	SW1	AB12	AC10	QN	LED2	AE7	AA7
J	SW_1	Y10	Y16				
K	SW_2	W10	AB15				
CK	SW_3	AA9	AA15				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ Switch의 상태를 [표 5-4]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

※ 주의: 표에서 "X" 표시일 경우 입력 상태를 '0'이나 '1'로 바꾸어 보며 출력의 변화 유무를 확인하고, 클록 펄스는 누름 스위치를 눌렀다가 떼는 것으로 누르면 '1', 떼면 '0'상태가 되는 것에 유의하며, "↓"는 클록 펄스의 하강 에지에서 플립플롭이 동작한다는 의미입니다.

〈표 6-29〉 결과 확인

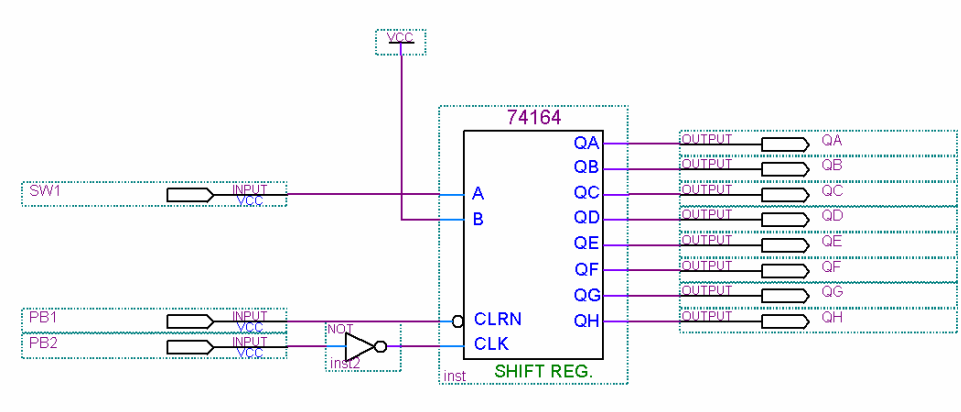
입 력					출 력	
CLR	PR	CK	J	K	Q	QN
0	1	X	X	X		
1	0	X	X	X		
1	1	↓	0	0		
		↓	0	1		
		↓	1	0		
		↓	1	1		

- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

6.6.3 실습 3 : 8 비트 시프트 레지스터

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

- ❑ New Project Wizard에서 Name을 EX_5_3으로 하여 프로젝트를 생성합니다.
- ❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-39]와 같이 논리 회로를 설계하고, EX_5_3라는 파일명으로 저장합니다.



[그림 6-39] 8비트 시프트 레지스터 회로도

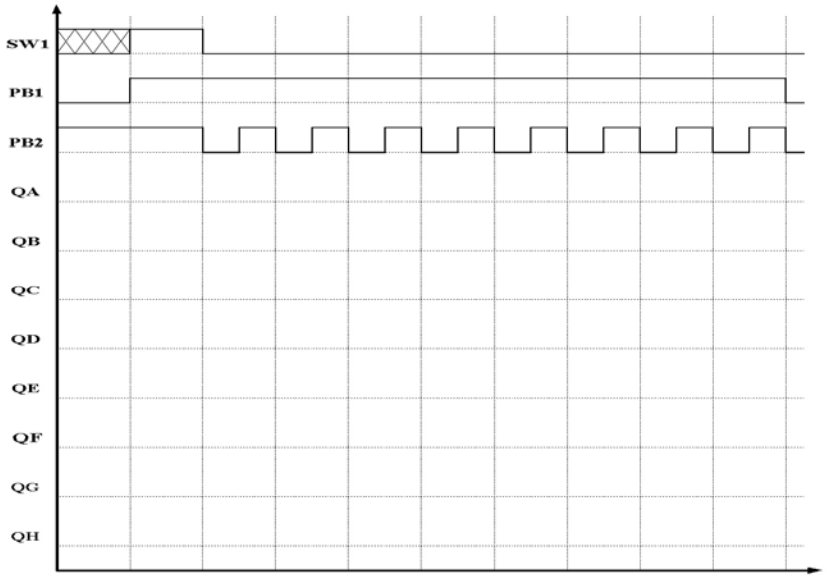
※ 참고 : VHDL 파일로 설계 EX_5_3_V.VHD 파일

```
ENTITY EX_5_3_V IS
    PORT(
        SW1, PB1, PB2          : IN BIT;    -- SW1 : A, PB1 : CLRN, PB2 : NOT CLK
        QA, QB, QC, QD         : OUT BIT;
        QE, QF, QG, QH         : OUT BIT);
END EX_5_3_V;

ARCHITECTURE HB OF EX_5_3_V IS
    SIGNAL TMP : BIT_VECTOR(0 TO 7);
BEGIN

PROCESS(SW1, PB1, PB2)
BEGIN
    IF PB1 = '0' THEN
        TMP <= "00000000";
    ELSIF PB2'EVENT AND PB2 = '1' THEN
        IF SW1 = '1' THEN
            TMP <= '1' & TMP(0 TO 6);
        ELSIF SW1 = '0' THEN
            TMP <= '0' & TMP(0 TO 6);
        END IF;
    END IF;
END PROCESS;
```

```
QA <= TMP(0);
QB <= TMP(1);
QC <= TMP(2);
QD <= TMP(3);
QE <= TMP(4);
QF <= TMP(5);
QG <= TMP(6);
QH <= TMP(7);

END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이번에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-40]에 기록합니다.



[그림 6-40] 결과 확인

2) 보드상에서의 확인

❑ [표 6-30]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-30〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
SW1	SW_1	Y10	AD10	QA	LED1	AF7	AB7
PB1	SW_2	W10	AC10	QB	LED2	AE7	AA7
PB2	SW_3	AA9	AA9	QC	LED3	AB8	AF7
				QD	LED4	W8	AC7
				QE	LED5	AF6	AD6
				QF	LED6	AE6	AC6
				QG	LED7	AD7	AF6
				QH	LED8	AC7	AE6

- ❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ 실험은 먼저 PB1을 눌렀다 떼어 모든 F/F를 리셋 시킵니다.
- ❑ 스위치 SW1을 꺼서 A 입력이 '1'이 되게 한 다음, 클록 스위치 PB2를 한번 눌렀다 놓는다. 이 때 QA 만 '1'이 됩니다.
- ❑ A 입력이 '0'이 되도록 SW1을 켭니다. 이 상태를 [표 6-31]의 초기값으로 기록하고, PB2를 한 번씩 눌렀다 뗀 다음 출력 상태를 관찰하여 표에 기록합니다.

〈표 6-31〉 결과 확인

클록 수	QA	QB	QC	QD	QE	QF	QG	QH
초기값								
1								
2								
3								
4								
5								
6								
7								
8								

- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

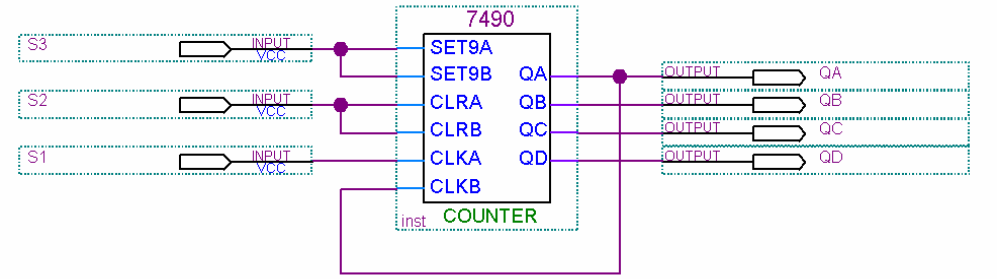
6.7 계수 회로

6.7.1 실습 1 : 비동기식 10진 계수기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_6_1으로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-41]과 같이 논리 회로를 설계하고, EX_6_1라는 파일명으로 저장합니다.



[그림 6-41] 비동기식 10진 계수기 회로도

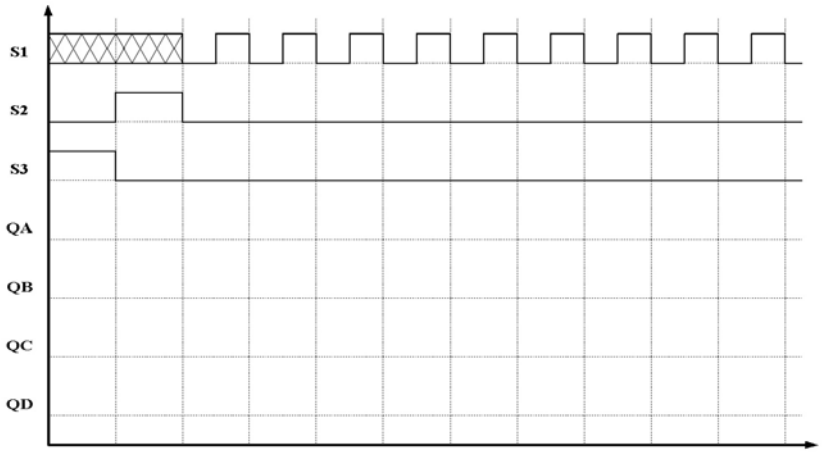
※ 참고 : VHDL 파일로 설계 EX_6_1_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY EX_6_1_V IS
    PORT(
        S3, S2, S1          : IN STD_LOGIC;       -- S3 : SET9, S2 : CLR, S1 : CLK
        QA, QB, QC, QD    : OUT STD_LOGIC );
END EX_6_1_V;

ARCHITECTURE HB OF EX_6_1_V IS
    SIGNAL TMP : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN

PROCESS(S3, S2, S1)
BEGIN
    IF S3 = '0' THEN
        TMP <= "1111";
    ELSIF S2 = '0' THEN
        TMP <= "0000";
    ELSIF S1'EVENT AND S1 = '1' THEN
        IF TMP = "1001" THEN
            TMP <= "0000";
        ELSE
```

```
                TMP <= TMP + 1;
            END IF;
        END IF;
    END PROCESS;

    QA <= TMP(3);
    QB <= TMP(2);
    QC <= TMP(1);
    QD <= TMP(0);

    END HB;
```

❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.

❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-42]에 기록합니다.



[그림 6-42] 결과 확인

2) 보드상에서의 확인

❑ [표 6-32]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-32〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
S1	SW_1	Y10	AD10	QA	LED1	AF7	AB7
S2	SW_2	W10	AC10	QB	LED2	AE7	AA7
S3	SW_3	AA9	AA9	QC	LED3	AB8	AF7
				QD	LED4	W8	AC7

- ❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- ❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.
- ❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- ❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- ❑ 각 스위치를 [표 6-33]와 같이 되도록 스위치를 변화시키며 출력을 관찰하여 표에 기록하고 위의 시뮬레이션 결과와 비교합니다.

〈표 6-33〉 결과 확인

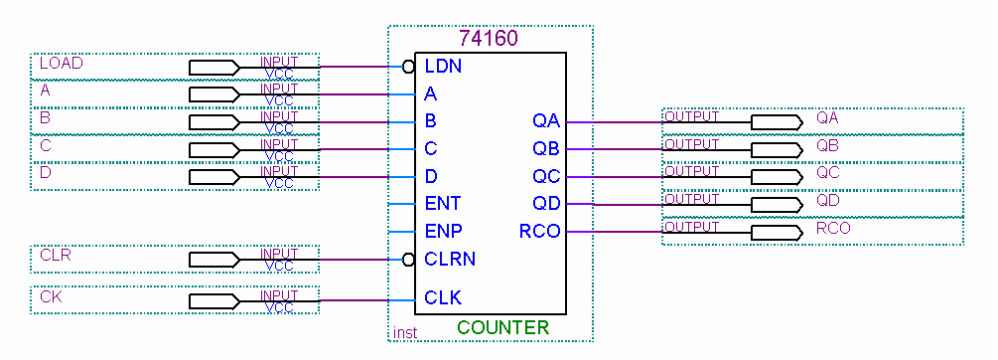
CLKA	CLR	SET	출 력				10 진
S1	S2	S3	QD	QC	QB	QA	
X	0	1					
X	1	0					
0	0	0					
1	0	0					
2	0	0					
3	0	0					
4	0	0					
5	0	0					
6	0	0					
7	0	0					
8	0	0					
9	0	0					

- ❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

6.7.2 실습 2 : 동기식 10진 계수기

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

- ❑ New Project Wizard에서 Name을 EX_6_2로 하여 프로젝트를 생성합니다.
- ❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-43]과 같이 논리 회로를 설계하고, EX_6_2라는 파일명으로 저장합니다.



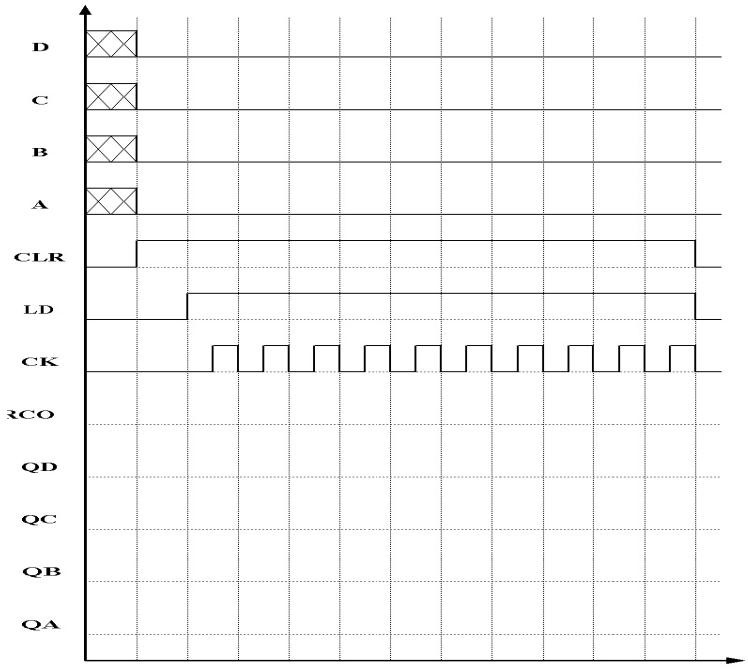
[그림 6-43] 동기식 10진 계수기 회로도

※ 참고 : VHDL 파일로 설계 EX_6_2_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY EX_6_2_V IS
    PORT(
        LOAD, CLR, CK      : IN STD_LOGIC;
        A, B, C, D         : IN STD_LOGIC;
        QA, QB, QC, QD     : OUT STD_LOGIC;
        RCO                : OUT STD_LOGIC);
END EX_6_2_V;

ARCHITECTURE HB OF EX_6_2_V IS
    SIGNAL TMP     : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN

PROCESS(LOAD, CLR, CK)
BEGIN
    IF CLR = '0' THEN
        TMP <= "0000";
        RCO <= '0';
    ELSIF CK'EVENT AND CK = '1' THEN
        IF LOAD = '0' THEN
            TMP <= A & B & C & D;
            RCO <= TMP(3) AND TMP(0);
```

```
            ELSE
                IF TMP = "1001" THEN
                    TMP <= "0000";
                    RCO <= '1';
                ELSE
                    TMP <= TMP + 1;
                    RCO <= '0';
                END IF;
            END IF;
        END IF;
    END PROCESS;

    QA <= TMP(3);
    QB <= TMP(2);
    QC <= TMP(1);
    QD <= TMP(0);

    END HB;
```

- ❑ Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.
- ❑ 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-44]에 기록합니다.



[그림 6-44] 결과 확인

2) 보드상에서의 확인

❑ [표 6-34]와 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-34〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
LOAD	SW0	Y13	Y16	QA	LED1	180	AB7
CLR	SW1	AB12	AB15	QB	LED2	179	AA7
A	SW_1	Y10	AD10	QC	LED3	178	AF7
B	SW_2	W10	AC10	QD	LED4	177	AC7
C	SW_3	AA9	AA9	RCO	LED6	176	AD6
D	SW_4	V9	Y9				
CK	SW_5	AE9	Y10				

❑ 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.

❑ Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.

❑ 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.

❑ 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.

❑ Switch의 상태를 [표 6-35]처럼 변화시키면서 LED에서의 출력을 표에 기록하고, 위의 시뮬레이션의 결과와 비교합니다.

〈표 6-35〉 결과 확인

초 기 값				클리어	로드	클록	출 력					
D	C	B	A	CLR	LOAD	CK	RCO	QD	QC	QB	QA	10진수
X	X	X	X	0	X	GND						
0	0	0	0	1	0	GND						
0	0	0	0	1	1	0						
						1						
						2						
						3						
						4						
						5						
						6						
						7						
						8						
						9						

❑ 모든 과정이 끝나면 전원을 끄고, 결과표의 입/출력 상태를 비교해 봅니다.

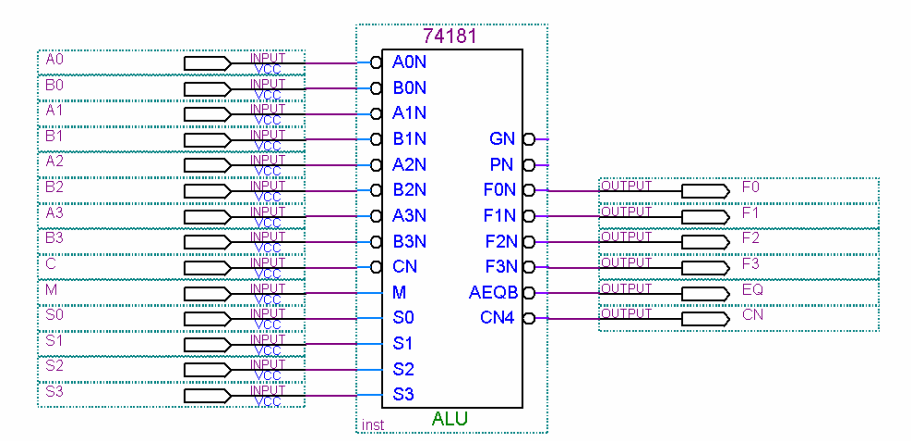
6.8 연산 논리 장치

6.8.1 실습 : 연산 논리 회로

1) Block Diagram/Schematic을 이용한 설계 및 시뮬레이션 검증

❑ New Project Wizard에서 Name을 EX_7_1로 하여 프로젝트를 생성합니다.

❑ 그래픽 설계 창을 열고, 심볼 라이브러리를 이용하여 [그림 6-45]와 같이 논리 회로를 설계하고, EX_7_1라는 파일명으로 저장합니다.



[그림 6-45] 연산 논리회로 회로도

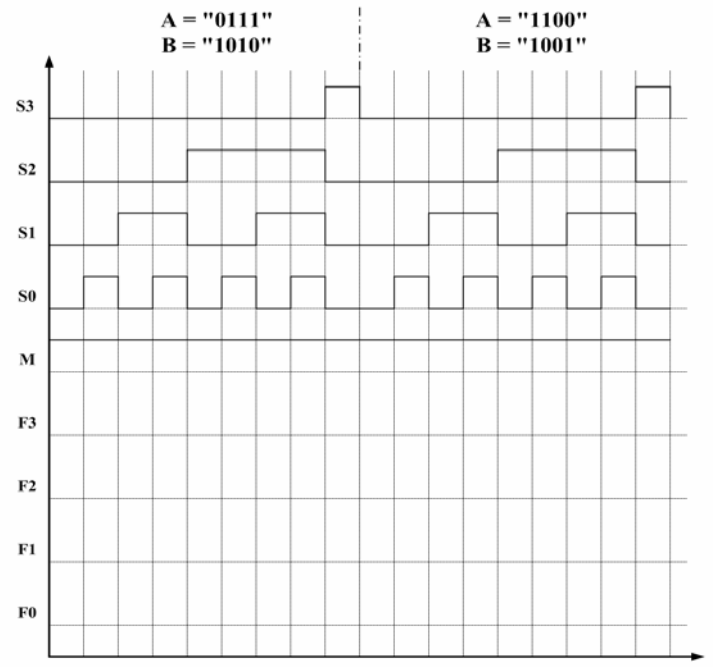
※ 참고 : VHDL 파일로 설계 EX_7_1_V.VHD 파일

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY EX_8_1_V IS
    PORT(
        M, C        : IN STD_LOGIC;
        S           : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        A, B        : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        EQ          : OUT STD_LOGIC;
        F           : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) );
END EX_8_1_V;

ARCHITECTURE HB OF EX_8_1_V IS
    SIGNAL TMP    : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN

PROCESS(S, A, B, M, C)
BEGIN
```

```
    IF M = '1' THEN
        CASE S IS
            WHEN "0000" =>
                TMP(3 DOWNTO 0) <= NOT A;
            WHEN "0001" =>
                TMP(3 DOWNTO 0) <= NOT ( A OR B);
            WHEN "0010" =>
                TMP(3 DOWNTO 0) <= ( NOT A ) AND B;
            WHEN "0011" =>
                TMP(3 DOWNTO 0) <= "0000";
            WHEN "0100" =>
                TMP(3 DOWNTO 0) <= NOT ( A AND B );
            WHEN "0101" =>
                TMP(3 DOWNTO 0) <= NOT B;
            WHEN "0110" =>
                TMP(3 DOWNTO 0) <= A XOR B;
            WHEN "0111" =>
                TMP(3 DOWNTO 0) <= A AND ( NOT B );
            WHEN "1000" =>
                TMP(3 DOWNTO 0) <= ( NOT A ) OR B;
            WHEN "1001" =>
                TMP(3 DOWNTO 0) <= NOT ( A XOR B );
            WHEN "1010" =>
                TMP(3 DOWNTO 0) <= B;
            WHEN "1011" =>
                TMP(3 DOWNTO 0) <= A AND B;
            WHEN "1100" =>
                TMP(3 DOWNTO 0) <= "1111";
            WHEN "1101" =>
                TMP(3 DOWNTO 0) <= A OR ( NOT B );
            WHEN "1110" =>
                TMP(3 DOWNTO 0) <= A OR B;
            WHEN "1111" =>
                TMP(3 DOWNTO 0) <= A;
            WHEN OTHERS =>
        END CASE;

    ELSIF M = '0' THEN
        CASE S IS
            WHEN "0000" =>
                TMP <= A;
            WHEN "0001" =>
                TMP <= ( A OR B );
            WHEN "0010" =>
                TMP <= ( A OR ( NOT B ));
            WHEN "0011" =>
                TMP <= "1111" ;
            WHEN "0100" =>
                TMP(3 DOWNTO 0) <= ( A + ( A AND ( NOT B ))) ;
            WHEN "0101" =>
                TMP(3 DOWNTO 0) <= (( A OR B ) + ( A AND ( NOT B ))) ;
            WHEN "0110" =>
                TMP(3 DOWNTO 0) <= A - B - "0001" ;
            WHEN "0111" =>
                TMP(3 DOWNTO 0) <= ( A AND ( NOT B )) - "0001" ;
            WHEN "1000" =>
                TMP(3 DOWNTO 0) <= A + ( A AND B ) ;
```

```
            WHEN "1001" =>
                TMP(3 DOWNTO 0) <= A + B ;
            WHEN "1010" =>
                TMP(3 DOWNTO 0) <= ( A OR ( NOT B )) + ( A AND B ) ;
            WHEN "1011" =>
                TMP(3 DOWNTO 0) <= ( A AND B ) - "0001" ;
            WHEN "1100" =>
                TMP(3 DOWNTO 0) <= A + A;
            WHEN "1101" =>
                TMP(3 DOWNTO 0) <= ( A OR B ) + A ;
            WHEN "1110" =>
                TMP(3 DOWNTO 0) <= ( A OR ( NOT B )) + A ;
            WHEN "1111" =>
                TMP(3 DOWNTO 0) <= A - "0001" ;
            WHEN OTHERS =>
        END CASE;
    END IF;
END PROCESS;

PROCESS(TMP, C, M)
BEGIN
    IF M = '1' OR C = '1' THEN
        F <= TMP;
    ELSIF C = '0' THEN
        F <= TMP + 1;
    END IF;
END PROCESS;

EQ <= '1' WHEN F= "1111" ELSE '0';

END HB;
```

- Compile 하여 문법 오류 검사 및 설계 파일에 대한 시뮬레이션 정보를 분석하게 됩니다.
- 시뮬레이션 검증 과정을 통해 이전에 설계한 회로에 대해 tool 내부에서 검증 하는 과정을 합니다. 시뮬레이션을 구동하고 그 결과를 [그림 6-46]에 기록합니다.



[그림 6-46] 결과 확인

2) 보드상에서의 확인

❑ [표 6-36]과 같이 각 입력 / 출력 포트의 핀 번호를 설정합니다.

〈표 6-36〉 핀 연결

입 력				출 력			
핀 이름	연결소자	Altera	Xilinx	핀 이름	연결소자	Altera	Xilinx
A0	SW0	Y13	Y16	F0	LED1	AF7	AB7
A1	SW1	AB12	AB15	F1	LED2	AE7	AA7
A2	SW2	AA12	AA15	F2	LED3	AB8	AF7
A3	SW3	AD12	AE15	F3	LED4	W8	AC7
B0	SW4	AC12	AD15	EQ	LED6	AF6	AD6
B1	SW5	U12	AF13	CN	LED7	AE6	AC6
B2	SW6	AE11	AA13				
B3	SW7	Y12	W15				
C	SW_1	Y10	AD10				
M	SW_2	W10	AC10				
S0	SW_3	AA9	AA9				
S1	SW_4	V9	Y9				
S2	SW_5	AE9	Y10				

S3	SW_6	AC9	AB9				

- 컴파일을 통해 핀 할당에 대한 정보를 프로젝트에 등록해 줍니다.
- Parallel port Cable을 JTAG 케이블과 연결하고, JTAG 케이블을 다시 디바이스 모듈에 연결합니다.
- 보드의 전원 커넥터를 연결하고 전원 스위치를 ON합니다.
- 설계 소프트웨어의 다운로드 창을 통해 디바이스에 프로그램 합니다.
- 논리 연산을 하기 위해 모드 M의 스위치를 '1' 상태로 놓고, 아래 표와 같이 선택 신호 입력을 변화시키며 출력을 관찰하여 [표 6-37]에 적습니다.

〈표 6-37〉 결과 확인 (1)

연산 선택 신호 입력				A = 0111, B = 1010				A = 1100, B = 1001				연산식
S3	S2	S1	S0	F3	F2	F1	F0	F3	F2	F1	F0	논리 기능
0	0	0	0									
0	0	0	1									
0	0	1	0									
0	0	1	1									
0	1	0	0									
0	1	0	1									
0	1	1	0									
0	1	1	1									
1	0	0	0									

- 표에서 입력에 따른 결과를 보고 어떤 논리 연산이 되었는가를 조사하여 표에 기록합니다.
- 논리 연산 과정에서 자리 올림 입력 C의 논리 레벨이 '0'과 '1'일 때의 결과에 따라서 차이가 있는가를 실험해 봅니다.
- 산술 연산을 하기 위해 모드 M을 '0'으로 하고, 아래 표와 같이 선택 신호 입력과 데이터 입력을 변화시키면서 출력을 관찰하여 표에 기록합니다.
- [표 7-3]에서 입력에 따른 결과를 보고 어떤 산술 연산이 되었는가를 조사하여 표에 10진수로 연산식을 써 넣습니다.

〈표 6-38〉 결과 확인 (2)

연산선택신호				데이터 입력											결과 출력						연산식
S3	S2	S1	S0	C	A3	A2	A1	A0	10진	B3	B2	B1	B0	10진	CN	F3	F2	F1	F0	A=B	
1	0	0	1	1	0	1	0	1		0	1	1	0								
					0	1	1	1		0	0	1	1								
				0	1	0	1	0		1	0	0	0								
0	1	1	0	0	1	0	1	0		0	1	0	1								
				1																	
				0	0	1	1	0		1	0	1	0								
				1																	
				1	0	0	1	1		0	0	1	1								
					0	1	1	1		0	0	1	1								

❑ 모든 과정이 끝나면 전원을 끄고, 산술 연산 과정에서 출력 A = B 단자는 어느 때 결과가 나오는지 살펴봅시다.

07
HBE-COMBO II User's Manual & Lab Guide

부 록

7. 부록

7.1 VFD 제어

HBE-COMBO II 보드 내에 장착된 16 X 2의 VFD(Vacuum Fluorescent Display)를 제어하기 위한 VHDL 예제입니다.

```
library ieee;
use ieee.std_logic_1164.all;

entity vfd_demo is
port(
        clk : in std_logic;                             -- 1kHz Clock
        rst         : in std_logic;

        vfd_e : out std_logic;
        vfd_rs : out std_logic;
        vfd_rw : out std_logic;
        vfd_data : out std_logic_vector(7 downto 0)
);
end vfd_demo;

architecture hb of vfd_demo is
        type state is (delay, function_set, entry_mode, disp_onoff, line1, line2, delay_t, clear_disp);
        signal vfd_state : state;
        signal cnt : integer range 0 to 4095;

        signal fun_cnt : integer range 0 to 1;
        signal vfd_fun : std_logic_vector(7 downto 0);

begin

-- 각 영역 별로 지연시간 정의
-- ex) cnt = 70 => 70 ms

process(rst, clk)
begin
        if rst = '1' then
                vfd_state <= delay;
```

```
	elsif clk'event and clk = '1' then
		case vfd_state is
			when delay =>
				if cnt = 70 then
					vfd_state <= function_set;
				end if;
			when function_set =>
				if cnt = 30 then
					vfd_state <= disp_onoff;
				end if;
			when disp_onoff =>
				if cnt = 30 then
					vfd_state <= entry_mode;
				end if;
			when entry_mode =>
				if cnt = 30 then
					vfd_state <= line1;
				end if;
			when line1 =>
				if cnt = 20 then
					vfd_state <= line2;
				end if;
			when line2 =>
				if cnt = 20 then
					vfd_state <= delay_t;
				end if;
			when delay_t =>
				if cnt = 4000 then
					vfd_state <= clear_disp;
				end if;
			when clear_disp =>
				if cnt = 1000 then
					vfd_state <= function_set;
				end if;
		end case;
	end if;
end process;

-- 각 영역별로 지연시간 카운트

process(rst, clk)
begin
	if rst = '1' then
		cnt <= 0;
	elsif clk'event and clk = '1' then
		case vfd_state is
			when delay =>
				if cnt = 70 then
					cnt <= 0;
				else
					cnt <= cnt + 1;
				end if;
			when function_set =>
				if cnt = 30 then
					cnt <= 0;
				else
					cnt <= cnt + 1;
```

```
                                        end if;
                                when disp_onoff =>
                                        if cnt = 30 then
                                                cnt <= 0;
                                        else
                                                cnt <= cnt + 1;
                                        end if;
                                when entry_mode =>
                                        if cnt = 30 then
                                                cnt <= 0;
                                        else
                                                cnt <= cnt + 1;
                                        end if;
                                when line1 =>
                                        if cnt = 20 then
                                                cnt <= 0;
                                        else
                                                cnt <= cnt + 1;
                                        end if;
                                when line2 =>
                                        if cnt = 20 then
                                                cnt <= 0;
                                        else
                                                cnt <= cnt + 1;
                                        end if;
                                when delay_t =>
                                        if cnt = 4000 then
                                                cnt <= 0;
                                        else
                                                cnt <= cnt + 1;
                                        end if;
                                when clear_disp =>
                                        if cnt = 1000 then
                                                cnt <= 0;
                                        else
                                                cnt <= cnt + 1;
                                        end if;
                        end case;
        end if;
end process;


process(rst, clk)
begin
        if rst = '1' then
                vfd_rs <= '0';
                vfd_rw <= '0';
                vfd_data <= "00000000";
        elsif clk'event and clk = '1' then
                case vfd_state is
                        when delay => null;
                    -- 밝기 조절을 위해 vfd_fun 변수 사용
                        when function_set =>
                                vfd_rs <= '0';
                                vfd_rw <= '0';
                                vfd_data <= vfd_fun;
                                fun_cnt <= fun_cnt + 1;
```

```
                when disp_onoff =>
                        vfd_rs <= '0';
                        vfd_rw <= '0';
                        vfd_data <= "00001100";
                when entry_mode =>
                        vfd_rs <= '0';
                        vfd_rw <= '0';
                        vfd_data <= "00000110";
                when clear_disp =>
                        vfd_rs <= '0';
                        vfd_rw <= '0';
                        vfd_data <= "00000001";
                when delay_t =>
                        vfd_rs <= '0';
                        vfd_rw <= '0';
                        vfd_data <= "00000010";
            -- 첫 번째 라인 문자 데이터 입력
                when line1 =>
                        vfd_rw <= '0';
                        case cnt is
                                when 0 =>
                                        vfd_rs <= '0';
                                        vfd_data <= "10000000";
                                when 1 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01001000";   -- H
                                when 2 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01000010";   -- B
                                when 3 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01000101";   -- E
                                when 4 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00101101";   -- -
                                when 5 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01000011";   -- C
                                when 6 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01101111";   -- o
                                when 7 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01101101";   -- m
                                when 8 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01100010";   -- b
                                when 9 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01101111";   -- o
                                when 10 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 11 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01001001";   -- I
                                when 12 =>
```

```
                                        vfd_rs <= '1';
                                        vfd_data <= "01001001";   -- I
                                when 13 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 14 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01000010";   -- B
                                when 15 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01100100";   -- d
                                when 16 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 17 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 18 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 19 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 20 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when others =>
                        end case;
                -- 두 번째 라인 문자 데이터 입력
                  when line2 =>
                        vfd_rw <= '0';
                        case cnt is
                                when 0 =>
                                        vfd_rs <= '0';
                                        vfd_data <= "11000000";
                                when 1 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01000100";   -- D
                                when 2 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01100101";   -- E
                                when 3 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01101101";   -- M
                                when 4 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01101111";   -- O
                                when 5 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "00100000";   --
                                when 6 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01100010";   -- b
                                when 7 =>
                                        vfd_rs <= '1';
                                        vfd_data <= "01111001";   -- y
                                when 8 =>
```

```
                                                vfd_rs <= '1';
                                                vfd_data <= "00100000";   --
                                        when 9 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01001000";   -- H
                                        when 10 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01000001";   -- A
                                        when 11 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01001110";   -- N
                                        when 12 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01000010";   -- B
                                        when 13 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01000001";   -- A
                                        when 14 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01000011";   -- C
                                        when 15 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "01001011";   -- K
                                        when 16 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "00101110";   -- .
                                        when 17 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "00100000";   --
                                        when 18 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "00100000";   --
                                        when 19 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "00100000";   --
                                        when 20 =>
                                                vfd_rs <= '1';
                                                vfd_data <= "00100000";   --
                                        when others =>
                                end case;
                        end case;
                end if;
end process;

-- VFD 밝기 조절을 위한 PROCESS 문
process(clk)

begin
if clk'event and clk = '1' then
        case fun_cnt is
                when 0 => vfd_fun <= "00111100";
                when 1 => vfd_fun <= "00111111";
                when others => vfd_fun <= "00111100";
        end case;
end if;
end process;
```

```
vfd_e <= clk when rst = '0' else '0';

end hb;
```

7.2 KEY PAD 제어

HBE-COMBO II 보드 내에 장착된 키 패드를 제어하여 스위치의 입력을 사용할 수 있는 예제입니다. 4X3의 12개의 버튼 스위치를 입력 받아서 사용할 수 있습니다. 예제는 키 패드를 스캔 방식을 이용하여 입력 받아 LED로 출력하는 예제입니다.

```
library ieee;
use ieee.std_logic_1164.all;

entity keypad_demo is
port(
        clk         : in std_logic;
        k_s         : buffer std_logic_vector(2 downto 0);
        k_d         : in std_logic_vector(3 downto 0);
        led         : out std_logic_vector(7 downto 0));
end keypad_demo;

architecture hb of keypad_demo is

signal scan         : integer range 0 to 2;
signal led_cnt      : integer range 0 to 15;
signal k_stop       : std_logic;
signal key_data     : integer range 0 to 15;

begin

-- 키 패드의 데이터 라인의 이벤트 검색(스캔 동작 결정을 위한 변수)
k_stop <= k_d(3) or k_d(2) or k_d(1) or k_d(0);

-- 캐 패드의 데이터 라인을 검색하여 이벤트가 발생하면 스캔 정지
process(clk)
begin
        if clk'event and clk = '1' then
                if k_stop = '1' then
                        null;
                else
                case k_s is
                        when "000" => key_data <= "001";
                        when "001" => key_data <= "010";
                        when "010" => key_data <= "100";
                        when "100" => key_data <= "001";
                        when others => null;
                end case;
                end if;
        end if;
end process;

-- 스캔에 따른 키 데이터 값을 검색하여 어떤 키 값이 입력 되었는지 검색
process(clk)
begin
        if clk'event and clk = '1' then
                case k_s is
```

```
                                when "001" =>
                                case k_d is
                                        when "0001" =>          key_data <= 1;
                                        when "0010" =>          key_data <= 4;
                                        when "0100" =>          key_data <= 7;
                                        when "1000" =>          key_data <= 0;
                                        when others =>          key_data <= 0;
                                end case;
                                when "010" =>
                                case k_d is
                                        when "0001" =>          key_data <= 2;
                                        when "0010" =>          key_data <= 5;
                                        when "0100" =>          key_data <= 8;
                                        when "1000" =>          key_data <= 10;
                                        when others =>          key_data <= 0;
                                end case;
                                when "100" =>
                                case k_d is
                                        when "0001" =>          key_data <= 3;
                                        when "0010" =>          key_data <= 6;
                                        when "0100" =>          key_data <= 9;
                                        when "1000" =>          key_data <= 0;
                                        when others =>          key_data <= 0;
                                end case;
                                when others => key_data <= 0;

                        end case;
                end if;
end process;

-- 키 값에 따른 LED에 데이터 출력
process(clk)
begin
if clk'event and clk = '1' then
                case key_data is
                        when 0      => led <= "00000000";
                        when 1      => led <= "00010001";
                        when 2      => led <= "00100010";
                        when 3      => led <= "00110011";
                        when 4      => led <= "01000100";
                        when 5      => led <= "01010101";
                        when 6      => led <= "01100110";
                        when 7      => led <= "01110111";
                        when 8      => led <= "10001000";
                        when 9      => led <= "10011001";
                        when 10     => led <= "10101010";
                        when 11     => led <= "10111011";
                        when 12     => led <= "11001100";
                        when 13     => led <= "11011101";
                        when 14     => led <= "11101110";
                        when 15     => led <= "11111111";
                end case;
end if;
end process;
end hb;
```

7.3 Segment 제어

HBE-COMBO II 보드 내에 장착된 7-Segment를 이용하여 시계를 구동하는 예제입니다. 6개의7-Segment를 사용하여 시, 분, 초의 시간을 표현할 수 있습니다. 하나의 탑 파일에서 2개의 하위 파일을 연동하여 사용하도록 구성되어 있습니다. 탑 파일에서는 시, 분, 초의 시계를 구동하는 기본 동작을 설계해 놓았습니다. 두 개의 하위 파일에서 시, 분, 초의 자릿수에 대한 숫자 정의를 위한 파일과, 7-Segment에 출력하기 위한 디코더용 파일을 정의해 놓았습니다.

```
-- file : watch.vhd

library ieee;
use ieee.std_logic_1164.all;

--port 선언
entity watch is
port(
        clk         : in std_logic;   -- 1KHz
        clear       : in std_logic;
        seg_com     : out std_logic_vector(7 downto 0);
        seg_data: out std_logic_vector(7 downto 0));
end watch;

architecture hb of watch is

-- 시계 구동에 사용하는 변수 선언
signal cnt                  : integer range 0 to 5;
signal num                  : integer range 0 to 15;
signal hour                 : integer range 0 to 59;
signal min, sec     : integer range 0 to 59;
signal h10, h1      : integer range 0 to 9;
signal m10, m1      : integer range 0 to 9;
signal s10, s1      : integer range 0 to 9;
signal cnts                 : integer range 0 to 499;
signal s_clk        : std_logic;
signal m_clk        : std_logic;
signal h_clk        : std_logic;

-- 하위 파일 연동
component sep
port(
        number      : in integer range 0 to 63;
        ten, one : out integer range 0 to 15);
end component;

component decod
port(
        bcd                 : in integer range 0 to 15;
        dot                 : in std_logic;
        seg_data: out std_logic_vector(7 downto 0));
end component;
```

```
-- 시, 분, 초 데이터 값 출력
begin
s_s       : sep
            port map(sec, s10, s1);
s_m       : sep
            port map(min, m10, m1);
s_h       : sep
            port map(hour, h10, h1);
-- 7-Segment 디코터
s7        : decod
            port map(num, s_clk, seg_data);

-- 초 단위로 클럭 분주
process(clk, clear)
begin
if clear = '1' then
            cnts <= 0;
            s_clk <= '0';
elsif clk'event and clk = '1' then
            if cnts = 499 then
                        cnts <= 0;
                        s_clk <= not s_clk;
            else
                        cnts <= cnts + 1;
            end if;
end if;
end process;

-- 초에 대한 데이터 값 정의
process(s_clk, clear)
begin
if clear = '1' then
            sec <= 0;
elsif s_clk'event and s_clk = '1' then
            if sec >= 59 then
                        m_clk <= '1';
                        sec <= 0;
            else
                        sec <= sec + 1;
                        m_clk <= '0';
            end if;
end if;
end process;

-- 분에 대한 데이터 값 정의
process(m_clk, clear)
begin
if clear = '1' then
            min <= 0;
elsif m_clk'event and m_clk = '1' then
            if min >= 59 then
                        h_clk <= '1';
                        min <= 0;
            else
                        min <= min + 1;
                        h_clk <= '0';
            end if;
```

```
end if;
end process;

-- 시간에 대한 데이터 값 정의
process(h_clk, clear)
begin
if clear = '1' then
        hour <= 0;
elsif h_clk'event and h_clk = '1' then
        if hour >= 23 then
                hour <= 0;
        else
                hour <= hour + 1;
        end if;
end if;
end process;

-- 7-Segment 출력
process(clk)
begin
if clk'event and clk = '1' then
        case cnt is
                when 0 =>
                        seg_com <= "01111111";
                        num <= h10;
                when 1 =>
                        seg_com <= "10111111";
                        num <= h10;
                when 2 =>
                        seg_com <= "11011111";
                        num <= h10;
                when 3 =>
                        seg_com <= "11101111";
                        num <= h10;
                when 4 =>
                        seg_com <= "11110111";
                        num <= h10;
                when 5 =>
                        seg_com <= "11111011";
                        num <= h10;
                when others =>
                        cnt <= 0;
        end case;
        cnt <= cnt + 1;
end if;
end process;
end hb;

-- File : sep.vhd
library ieee;
use ieee.std_logic_1164.all;

entity sep is
port(
        number     : in integer range 0 to 59;
        ten, one: out integer range 0 to 9);
end sep;
```

```
architecture hb of sep is
begin

-- 시, 분, 초의 데이터 값을 받아들여 자릿수를 나누어 출력
process(number)
begin
if number <= 9 then
        ten <= 0;
        one <= number;
elsif number <= 19 then
        ten <= 1;
        one <= number - 10;
elsif number <= 29 then
        ten <= 2;
        one <= number - 20;
elsif number <= 39 then
        ten <= 3;
        one <= number - 30;
elsif number <= 49 then
        ten <= 4;
        one <= number - 40;
elsif number <= 59 then
        ten <= 5;
        one <= number - 50;
else
        ten <= 0;
        one <= 0;
end if;
end process;
end hb;

library ieee;
use ieee.std_logic_1164.all;

File : decod.vhd
entity decod is
port(
        bcd                     : in integer range 0 to 15;
        dot                     : in std_logic;
        seg_data: out std_logic_vector(7 downto 0));
end decod;

architecture hb of decod is

signal seg_d        : std_logic_vector(6 downto 0);

begin

-- 7-Segment 데이터 값으로 변환
process(bcd)
begin
case bcd is
        when 0 => seg_d <= "1111110";
        when 1 => seg_d <= "0110000";
        when 2 => seg_d <= "1101101";
        when 3 => seg_d <= "1111001";
```

```
            when 4 => seg_d <= "0110011";
            when 5 => seg_d <= "1011011";
            when 6 => seg_d <= "1011111";
            when 7 => seg_d <= "1110000";
            when 8 => seg_d <= "1111111";
            when 9 => seg_d <= "1110011";
            when others => seg_d <= "0000000";
end case;
seg_data <= seg_d & dot;
end process;
end hb;
```

7.4 DOT-Matrix 제어

HBE-COMBO II 보드 내에 장착된 14 X 10의 Dot-LED를 이용하여 문자를 표시 할 수 있는 장치입니다. 여기 예제에서는 스캔 방식을 이용하여 DOT의 LED를 제어하는 방식을 사용하고 있습니다. 또한 2개의 설계 파일을 이용하여 탑 파일에서 하위 파일을 component 문으로 서로 연동하여 사용하고 있습니다.

탑 파일에서는 DOT를 표시하는 데이터의 값을 정의하고, 그 데이터 값을 시프트 하는 방식으로 글자가 한 칸이 이동하는 동작 상태를 정의해 놓은 것입니다. 하위 파일은 스캔으로 구동시키는 역할을 하고 있습니다. 따라서 스캔에 따라 라인을 선택하고 그 라인에 맞는 데이터 값을 정의해 주는 동작을 하고 있습니다. 밑에 예제에서는 2개의 파일을 모두 보여주고 있습니다.

```
-- file : dot_matrix.vhd
library ieee;
use ieee.std_logic_1164.all;

entity dot_matrix is
port (
        clk : in std_logic;   -- 1kHz

        dot_d : out std_logic_vector ( 13 downto 0);
        dot_scan : out std_logic_vector ( 9 downto 0)
);
end dot_matrix;

architecture hb of dot_matrix is

-- 하위 파일 정의
component dot_disp
port (
        clk : in std_logic;
        dot_data_00 : in std_logic_vector (13 downto 0);
        dot_data_01 : in std_logic_vector (13 downto 0);
        dot_data_02 : in std_logic_vector (13 downto 0);
        dot_data_03 : in std_logic_vector (13 downto 0);
        dot_data_04 : in std_logic_vector (13 downto 0);
        dot_data_05 : in std_logic_vector (13 downto 0);
        dot_data_06 : in std_logic_vector (13 downto 0);
        dot_data_07 : in std_logic_vector (13 downto 0);
        dot_data_08 : in std_logic_vector (13 downto 0);
        dot_data_09 : in std_logic_vector (13 downto 0);

        dot_d : out std_logic_vector (13 downto 0);
        dot_scan : out std_logic_vector ( 9 downto 0)
);
end component;

-- 스캔 및 데이터 라인에 출력 데이터 동작을 위한 변수 선언
signal dot_data_00 : std_logic_vector (13 downto 0);
```

```
    signal dot_data_01 : std_logic_vector (13 downto 0);
    signal dot_data_02 : std_logic_vector (13 downto 0);
    signal dot_data_03 : std_logic_vector (13 downto 0);
    signal dot_data_04 : std_logic_vector (13 downto 0);
    signal dot_data_05 : std_logic_vector (13 downto 0);
    signal dot_data_06 : std_logic_vector (13 downto 0);
    signal dot_data_07 : std_logic_vector (13 downto 0);
    signal dot_data_08 : std_logic_vector (13 downto 0);
    signal dot_data_09 : std_logic_vector (13 downto 0);

    signal data_00 : std_logic_vector (13 downto 0);
    signal data_01 : std_logic_vector (13 downto 0);
    signal data_02 : std_logic_vector (13 downto 0);
    signal data_03 : std_logic_vector (13 downto 0);
    signal data_04 : std_logic_vector (13 downto 0);
    signal data_05 : std_logic_vector (13 downto 0);
    signal data_06 : std_logic_vector (13 downto 0);
    signal data_07 : std_logic_vector (13 downto 0);
    signal data_08 : std_logic_vector (13 downto 0);
    signal data_09 : std_logic_vector (13 downto 0);
    signal data_10 : std_logic_vector (13 downto 0);
    signal data_11 : std_logic_vector (13 downto 0);
    signal data_12 : std_logic_vector (13 downto 0);
    signal data_13 : std_logic_vector (13 downto 0);
    signal data_14 : std_logic_vector (13 downto 0);
    signal data_15 : std_logic_vector (13 downto 0);
    signal data_16 : std_logic_vector (13 downto 0);
    signal data_17 : std_logic_vector (13 downto 0);
    signal data_18 : std_logic_vector (13 downto 0);
    signal data_19 : std_logic_vector (13 downto 0);
    signal data_20 : std_logic_vector (13 downto 0);
    signal data_21 : std_logic_vector (13 downto 0);
    signal data_22 : std_logic_vector (13 downto 0);
    signal data_23 : std_logic_vector (13 downto 0);
    signal data_24 : std_logic_vector (13 downto 0);
    signal data_25 : std_logic_vector (13 downto 0);
    signal data_26 : std_logic_vector (13 downto 0);
    signal data_27 : std_logic_vector (13 downto 0);
    signal data_28 : std_logic_vector (13 downto 0);
    signal data_29 : std_logic_vector (13 downto 0);
    signal data_30 : std_logic_vector (13 downto 0);
    signal data_31 : std_logic_vector (13 downto 0);
    signal data_32 : std_logic_vector (13 downto 0);
    signal data_33 : std_logic_vector (13 downto 0);
    signal data_34 : std_logic_vector (13 downto 0);
    signal data_35 : std_logic_vector (13 downto 0);
    signal data_36 : std_logic_vector (13 downto 0);
    signal data_37 : std_logic_vector (13 downto 0);
    signal data_38 : std_logic_vector (13 downto 0);
    signal data_39 : std_logic_vector (13 downto 0);
    signal data_40 : std_logic_vector (13 downto 0);
    signal data_41 : std_logic_vector (13 downto 0);
    signal data_42 : std_logic_vector (13 downto 0);
    signal data_43 : std_logic_vector (13 downto 0);
    signal data_44 : std_logic_vector (13 downto 0);
    signal data_45 : std_logic_vector (13 downto 0);
    signal data_46 : std_logic_vector (13 downto 0);
```

```
signal data_47 : std_logic_vector (13 downto 0);
signal data_48 : std_logic_vector (13 downto 0);
signal data_49 : std_logic_vector (13 downto 0);
signal data_50 : std_logic_vector (13 downto 0);
signal data_51 : std_logic_vector (13 downto 0);

signal cnt_clk : integer range 24 downto 0;
signal cnt_data : integer range 51 downto 0;
signal clk_20h : std_logic;

begin
-- 출력할 코드 선언
data_00 <= "00000001100010";
data_01 <= "11110010010010";
data_02 <= "10000100001010";
data_03 <= "10000100001011";
data_04 <= "10000100001010";
data_05 <= "10000010010010";
data_06 <= "10000001100010";
data_07 <= "10000000000000";
data_08 <= "10001111111110";
data_09 <= "10000000100000";
data_10 <= "00000000000000";
data_11 <= "00000011111110";
data_12 <= "00010010010000";
data_13 <= "00010010010000";
data_14 <= "00010010010000";
data_15 <= "00010010010000";
data_16 <= "00010011111110";
data_17 <= "00010000000000";
data_18 <= "00010011111110";
data_19 <= "00010000010000";
data_20 <= "11110011111110";
data_21 <= "00000000000000";
data_22 <= "00000010000010";
data_23 <= "11110001000010";
data_24 <= "10000000100010";
data_25 <= "10000000011110";
data_26 <= "10000000100010";
data_27 <= "10000001000010";
data_28 <= "10000010000010";
data_29 <= "10000000100000";
data_30 <= "10001111111110";
data_31 <= "10000000000000";
data_32 <= "00000000000000";
data_33 <= "00000010000010";
data_34 <= "00000001000010";
data_35 <= "00000000100010";
data_36 <= "00000000011110";
data_37 <= "00000000100010";
data_38 <= "00000001000010";
data_39 <= "00000010000010";
data_40 <= "00000000000000";
data_41 <= "11111111111110";
data_42 <= "00000000100000";
data_43 <= "00000000000000";
data_44 <= "00000000000000";
```

```
data_45 <= "00000000000000";
data_46 <= "00000000000000";
data_47 <= "00000000000000";
data_48 <= "00000000000000";
data_49 <= "00000000000000";
data_50 <= "00000000000000";
data_51 <= "00000000000000";

-- dot 구동을 위한 클럭 분주
process (clk)
begin
	if clk'event and clk = '1' then
		if cnt_clk = 24 then
			cnt_clk <= 0;
			clk_20h <= not clk_20h;
		else
			cnt_clk <= cnt_clk + 1;
			clk_20h <= clk_20h;
		end if;
	end if;
end process;

process (clk_20h)
begin
	if clk_20h'event and clk_20h = '1' then
		if cnt_data = 51 then
			cnt_data <= 0;
		else
			cnt_data <= cnt_data + 1;
		end if;
	end if;
end process;

-- dot_data_09에 사전에 정의된 데이터 시프트 하면서 넣어주는 프로세서문
process (clk_20h)
begin
	if clk_20h'event and clk_20h = '1' then
		dot_data_00 <= dot_data_01;
		dot_data_01 <= dot_data_02;
		dot_data_02 <= dot_data_03;
		dot_data_03 <= dot_data_04;
		dot_data_04 <= dot_data_05;
		dot_data_05 <= dot_data_06;
		dot_data_06 <= dot_data_07;
		dot_data_07 <= dot_data_08;
		dot_data_08 <= dot_data_09;
		case cnt_data is
			when 0 =>
				dot_data_09 <= data_00;
			when 1 =>
				dot_data_09 <= data_01;
			when 2 =>
				dot_data_09 <= data_02;
			when 3 =>
				dot_data_09 <= data_03;
			when 4 =>
				dot_data_09 <= data_04;
```

```
when 5 =>
        dot_data_09 <= data_05;
when 6 =>
        dot_data_09 <= data_06;
when 7 =>
        dot_data_09 <= data_07;
when 8 =>
        dot_data_09 <= data_08;
when 9 =>
        dot_data_09 <= data_09;
when 10 =>
        dot_data_09 <= data_10;
when 11 =>
        dot_data_09 <= data_11;
when 12 =>
        dot_data_09 <= data_12;
when 13 =>
        dot_data_09 <= data_13;
when 14 =>
        dot_data_09 <= data_14;
when 15 =>
        dot_data_09 <= data_15;
when 16 =>
        dot_data_09 <= data_16;
when 17 =>
        dot_data_09 <= data_17;
when 18 =>
        dot_data_09 <= data_18;
when 19 =>
        dot_data_09 <= data_19;
when 20 =>
        dot_data_09 <= data_20;
when 21 =>
        dot_data_09 <= data_21;
when 22 =>
        dot_data_09 <= data_22;
when 23 =>
        dot_data_09 <= data_23;
when 24 =>
        dot_data_09 <= data_24;
when 25 =>
        dot_data_09 <= data_25;
when 26 =>
        dot_data_09 <= data_26;
when 27 =>
        dot_data_09 <= data_27;
when 28 =>
        dot_data_09 <= data_28;
when 29 =>
        dot_data_09 <= data_29;
when 30 =>
        dot_data_09 <= data_30;
when 31 =>
        dot_data_09 <= data_31;
when 32 =>
        dot_data_09 <= data_32;
when 33 =>
```

```
                                        dot_data_09 <= data_33;
                                when 34 =>
                                        dot_data_09 <= data_34;
                                when 35 =>
                                        dot_data_09 <= data_35;
                                when 36 =>
                                        dot_data_09 <= data_36;
                                when 37 =>
                                        dot_data_09 <= data_37;
                                when 38 =>
                                        dot_data_09 <= data_38;
                                when 39 =>
                                        dot_data_09 <= data_39;
                                when 40 =>
                                        dot_data_09 <= data_40;
                                when 41 =>
                                        dot_data_09 <= data_41;
                                when 42 =>
                                        dot_data_09 <= data_42;
                                when 43 =>
                                        dot_data_09 <= data_43;
                                when 44 =>
                                        dot_data_09 <= data_44;
                                when 45 =>
                                        dot_data_09 <= data_45;
                                when 46 =>
                                        dot_data_09 <= data_46;
                                when 47 =>
                                        dot_data_09 <= data_47;
                                when 48 =>
                                        dot_data_09 <= data_48;
                                when 49 =>
                                        dot_data_09 <= data_49;
                                when 50 =>
                                        dot_data_09 <= data_50;
                                when 51 =>
                                        dot_data_09 <= data_51;
                        end case;
        end if;
end process;

-- 하위 파일을 직접 변수를 사용하여 상위 파일과 연동
u0 : dot_disp
port map (
        clk => clk,
        dot_data_00 => dot_data_00,
        dot_data_01 => dot_data_01,
        dot_data_02 => dot_data_02,
        dot_data_03 => dot_data_03,
        dot_data_04 => dot_data_04,
        dot_data_05 => dot_data_05,
        dot_data_06 => dot_data_06,
        dot_data_07 => dot_data_07,
        dot_data_08 => dot_data_08,
        dot_data_09 => dot_data_09,
        dot_d => dot_d,
        dot_scan => dot_scan
```

```
);
end hb;


-- file : dot_disp.vhd
library ieee;
use ieee.std_logic_1164.all;

entity dot_disp is
port (
        clk             : in std_logic;
        dot_data_00 : in std_logic_vector (13 downto 0);
        dot_data_01 : in std_logic_vector (13 downto 0);
        dot_data_02 : in std_logic_vector (13 downto 0);
        dot_data_03 : in std_logic_vector (13 downto 0);
        dot_data_04 : in std_logic_vector (13 downto 0);
        dot_data_05 : in std_logic_vector (13 downto 0);
        dot_data_06 : in std_logic_vector (13 downto 0);
        dot_data_07 : in std_logic_vector (13 downto 0);
        dot_data_08 : in std_logic_vector (13 downto 0);
        dot_data_09 : in std_logic_vector (13 downto 0);

        dot_d : out std_logic_vector (13 downto 0);
        dot_scan : out std_logic_vector ( 9 downto 0)
);
end dot_disp;

architecture hb of dot_disp is
signal cnt_clk : integer range 9 downto 0;
begin

-- 스캔을 위한 cnt 값을 정의
process(clk)
begin
        if clk'event and clk = '1' then
                if cnt_clk = 9 then
                        cnt_clk <= 0;
                else
                        cnt_clk <= cnt_clk + 1;
                end if;
        end if;
end process;

-- 스캔 구동, 10개의 스캔 라인을 하나 씩 선택하면서 해당되는 데이터 값 입력
process(cnt_clk)
begin
        case cnt_clk is
                when 0 =>
                        dot_d <= dot_data_00;
                        dot_scan <= "0000000001";
                when 1 =>
                        dot_d <= dot_data_01;
                        dot_scan <= "0000000010";
                when 2 =>
                        dot_d <= dot_data_02;
                        dot_scan <= "0000000100";
                when 3 =>
```

```
                                dot_d <= dot_data_03;
                                dot_scan <= "0000001000";
                        when 4 =>
                                dot_d <= dot_data_04;
                                dot_scan <= "0000010000";
                        when 5 =>
                                dot_d <= dot_data_05;
                                dot_scan <= "0000100000";
                        when 6 =>
                                dot_d <= dot_data_06;
                                dot_scan <= "0001000000";
                        when 7 =>
                                dot_d <= dot_data_07;
                                dot_scan <= "0010000000";
                        when 8 =>
                                dot_d <= dot_data_08;
                                dot_scan <= "0100000000";
                        when 9 =>
                                dot_d <= dot_data_09;
                                dot_scan <= "1000000000";
        end case;
end process;

end a;
```

7.5 시리얼 통신

HBE-COMBO II 보드 내에 장착된 UART Port나 USB to Serial port를 이용하여 시리얼 통신 예제를 구동할 수 있습니다. 여기에서는 25MHz의 클럭을 입력받아 115200의 전송속도로 조절하여 PC의 데이터를 장비와 통신 할 수 있는 예제를 보여주고 있습니다.

장비에 현재 소스를 다운한 후 하이퍼터미널의 통신용 프로그램을 이용하여 장비의 데이터 전송을 할 수 있습니다. 이 예제는 장비에 있는 9핀 UART 케이블과 USB 케이블을 이용하여 본 예제를 구동할 수 있습니다. USB 포트를 이용할 경우 USB 전용 칩을 이용하기 때문에 디바이스 전용 드라이브를 설치 하여야 합니다. 이 드라이브에 대한 설명을 매뉴얼을 참고하기 바랍니다.

```
-- file : rs232_demo.vhd
-- 시리얼 통신 예제의 top 파일
-- 클럭에 따른 데이터를 시리얼로 전송
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rs232_demo is
        port (
                reset : in std_logic;
                clk : in std_logic;                     -- 25MHz input clock

                parity_en : in std_logic;               -- bus switch
                parity_mode : in std_logic;             -- odd : 1, even : 0, bus switch

                rxd : in std_logic;
                txd : out std_logic
        );
end rs232_demo;

architecture hb of rs232_demo is

component rs_232 is
        port (
                reset : in std_logic;
                clk : in std_logic;

                parity_en : in std_logic;
                parity_mode : in std_logic;

                rxd : in std_logic;
                txd : out std_logic;

                rxd_en : out std_logic;
                txd_en : in std_logic;
                uart_data_in : out std_logic_vector(7 downto 0);
                uart_data_out : in std_logic_vector(7 downto 0)
        );
```

```
end component;

signal resetn : std_logic;

signal rxd_en : std_logic;
signal txd_en : std_logic;
signal uart_data_in : std_logic_vector(7 downto 0);
signal uart_data_out : std_logic_vector(7 downto 0);

signal reg_rxd_en : std_logic_vector(7 downto 0);
signal reg_data_in0 : std_logic_vector(7 downto 0);
signal reg_data_in1 : std_logic_vector(7 downto 0);
signal reg_data_in2 : std_logic_vector(7 downto 0);
signal reg_data_in3 : std_logic_vector(7 downto 0);
signal reg_data_in4 : std_logic_vector(7 downto 0);
signal reg_data_in5 : std_logic_vector(7 downto 0);
signal reg_data_in6 : std_logic_vector(7 downto 0);
signal reg_data_in7 : std_logic_vector(7 downto 0);

begin

resetn <= not reset;


u_rs_232 : rs_232
        port map (
                reset => reset,
                clk => clk,

                parity_en => parity_en,
                parity_mode => parity_mode,
                rxd => rxd,
                txd => txd,

                rxd_en => rxd_en,
                txd_en => txd_en,

                uart_data_in => uart_data_in,
                uart_data_out => uart_data_out
        );

process (resetn, clk)
begin
        if resetn = '0' then
                reg_rxd_en <= (others => '0');
        elsif clk'event and clk = '1' then
                reg_rxd_en <= reg_rxd_en(6 downto 0) & rxd_en;
        end if;
end process;

txd_en <= reg_rxd_en(7);

process (resetn, clk)
begin
        if resetn = '0' then
                reg_data_in0 <= (others => '0');
                reg_data_in1 <= (others => '0');
```

```
                        reg_data_in2 <= (others => '0');
                        reg_data_in3 <= (others => '0');
                        reg_data_in4 <= (others => '0');
                        reg_data_in5 <= (others => '0');
                        reg_data_in6 <= (others => '0');
                        reg_data_in7 <= (others => '0');
        elsif clk'event and clk = '1' then
                        reg_data_in0 <= uart_data_in;
                        reg_data_in1 <= reg_data_in0;
                        reg_data_in2 <= reg_data_in1;
                        reg_data_in3 <= reg_data_in2;
                        reg_data_in4 <= reg_data_in3;
                        reg_data_in5 <= reg_data_in4;
                        reg_data_in6 <= reg_data_in5;
                        reg_data_in7 <= reg_data_in6;
        end if;
end process;

uart_data_out <= reg_data_in7;


end hb;

-- file : rs_232.vhd
-- 데이터 전송 속도 설정

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rs_232 is
        port (
                        reset : in std_logic;
                        clk : in std_logic;    -- 25MHz input clock

                        parity_en : in std_logic;
                        parity_mode : in std_logic;

                        rxd : in std_logic;
                        txd : out std_logic;

                        rxd_en : out std_logic;
                        txd_en : in std_logic;
                        uart_data_in : out std_logic_vector(7 downto 0);
                        uart_data_out : in std_logic_vector(7 downto 0)
        );
end rs_232;

architecture hb of rs_232 is

signal resetn : std_logic := '0';

constant bit_rate : std_logic_vector(7 downto 0) := "10101101";    -- 20M / 115200 = 173
constant bit_rate_half : std_logic_vector(7 downto 0) := "01010110";

type rx_state is (rx_idle, rx_start, data_rx, rx_parity, rx_stop);
signal rx_routine : rx_state;
```

```
type tx_state is (tx_idle, tx_start, data_tx, tx_parity, tx_stop);
signal tx_routine : tx_state;

signal rx_bit_add : std_logic_vector(7 downto 0);

signal cnt_rx_bit_rate : std_logic_vector(7 downto 0);
signal cnt_rx_data : std_logic_vector(2 downto 0);
signal cnt_tx_bit_rate : std_logic_vector(7 downto 0);
signal cnt_tx_data : std_logic_vector(2 downto 0);

signal rx_data : std_logic_vector(7 downto 0);
signal tx_data : std_logic_vector(7 downto 0);

signal reg_rxd : std_logic_vector(15 downto 0);

signal tx_parity_chk : std_logic;

begin

resetn <= not reset;

process (resetn, clk)
begin
	if resetn = '0' then
		reg_rxd <= (others => '0');
	elsif clk'event and clk = '1' then
		reg_rxd <= rxd & reg_rxd(15 downto 1);
	end if;
end process;


process (resetn, clk)
begin
	if resetn = '0' then
		rx_bit_add <= (others => '0');
	elsif clk'event and clk = '1' then
		if rx_routine = data_rx then
			if cnt_rx_bit_rate = bit_rate then
				rx_bit_add <= (others => '0');
			else
				rx_bit_add <= rx_bit_add + reg_rxd(0);
			end if;
		else
			rx_bit_add <= (others => '0');
		end if;
	end if;
end process;

process (resetn, clk)
begin
	if resetn = '0' then
		rx_routine <= rx_idle;
	elsif clk'event and clk = '1' then
		case rx_routine is
			when rx_idle =>
				if reg_rxd = "0000000000000000" then
```

```
                                        rx_routine <= rx_start;
                                end if;
                        when rx_start =>
                                if cnt_rx_bit_rate = bit_rate then
                                        rx_routine <= data_rx;
                                elsif cnt_rx_bit_rate = bit_rate_half then
                                        if reg_rxd(0) = '1' then
                                                rx_routine <= rx_idle;
                                        end if;
                                end if;
                        when data_rx =>
                                if cnt_rx_data = "111" then
                                        if cnt_rx_bit_rate = bit_rate then
                                                if parity_en = '1' then
                                                        rx_routine <= rx_parity;
                                                else
                                                        rx_routine <= rx_stop;
                                                end if;
                                        end if;
                                end if;
                        when rx_parity =>
                                if cnt_rx_bit_rate = bit_rate then
                                        rx_routine <= rx_stop;
                                end if;
                        when rx_stop =>
                                if cnt_rx_bit_rate(5 downto 0) = bit_rate(7 downto 2) then
                                        rx_routine <= rx_idle;
                                end if;
                end case;
        end if;
end process;



process (resetn, clk)
begin
        if resetn = '0' then
                cnt_rx_bit_rate <= (others => '0');
        elsif clk'event and clk = '1' then
                if rx_routine = rx_idle then
                        cnt_rx_bit_rate <= (others => '0');
                elsif cnt_rx_bit_rate = bit_rate then
                        cnt_rx_bit_rate <= (others => '0');
                else
                        cnt_rx_bit_rate <= cnt_rx_bit_rate + '1';
                end if;
        end if;
end process;



process (resetn, clk)
begin
        if resetn = '0' then
                cnt_rx_data <= (others => '0');
        elsif clk'event and clk = '1' then
                if rx_routine = data_rx then
```

```
                                        if cnt_rx_bit_rate = bit_rate then
                                                if cnt_rx_data = "111" then
                                                        cnt_rx_data <= (others => '0');
                                                else
                                                        cnt_rx_data <= cnt_rx_data + '1';
                                                end if;
                                        end if;
                        else
                                        cnt_rx_data <= (others => '0');
                        end if;
        end if;
end process;

process (resetn, clk)
begin
        if resetn = '0' then
                        rx_data <= (others => '0');
        elsif clk'event and clk = '1' then
                        if rx_routine = data_rx then
                                        if cnt_rx_bit_rate = bit_rate - 1 then
                                                if rx_bit_add >= bit_rate_half then
                                                        rx_data <= '1' & rx_data(7 downto 1);
                                                else
                                                        rx_data <= '0' & rx_data(7 downto 1);
                                                end if;
                                        end if;
                        elsif rx_routine = rx_idle then
                                        rx_data <= (others => '0');
                        end if;
        end if;
end process;

process (resetn, clk)
begin
        if resetn = '0' then
                        rxd_en <= '0';
        elsif clk'event and clk = '1' then
                        if rx_routine = rx_stop then
                                        if cnt_rx_bit_rate >=   10 then
                                                rxd_en <= '1';
                                        else
                                                rxd_en <= '0';
                                        end if;
                        else
                                        rxd_en <= '0';
                        end if;
        end if;
end process;

process (resetn, clk)
begin
        if resetn = '0' then
                        uart_data_in <= (others => '0');
        elsif clk'event and clk = '1' then
                        if rx_routine = rx_stop then
                                        uart_data_in <= rx_data;
                        end if;
```

```
        end if;
end process;



process (resetn, clk)
begin
        if resetn = '0' then
                tx_routine <= tx_idle;
        elsif clk'event and clk = '1' then
                case tx_routine is
                        when tx_idle =>
                                if txd_en = '1' then
                                        tx_routine <= tx_start;
                                end if;
                        when tx_start =>
                                if cnt_tx_bit_rate = bit_rate then
                                        tx_routine <= data_tx;
                                end if;
                        when data_tx =>
                                if cnt_tx_data = "111" then
                                        if cnt_tx_bit_rate = bit_rate then
                                                if parity_en = '1' then
                                                        tx_routine <= tx_parity;
                                                else
                                                        tx_routine <= tx_stop;
                                                end if;
                                        end if;
                                end if;
                        when tx_parity =>
                                if cnt_tx_bit_rate = bit_rate then
                                        tx_routine <= tx_stop;
                                end if;
                        when tx_stop =>
                                if cnt_tx_bit_rate = bit_rate - 10 then
                                        tx_routine <= tx_idle;
                                end if;
                        when others =>
                                tx_routine <= tx_idle;
                end case;
        end if;
end process;

process (resetn, clk)
begin
        if resetn = '0' then
                cnt_tx_bit_rate <= (others => '0');
        elsif clk'event and clk = '1' then
                if tx_routine = tx_idle then
                        cnt_tx_bit_rate <= (others => '0');
                elsif cnt_tx_bit_rate = bit_rate then
                        cnt_tx_bit_rate <= (others => '0');
                else
                        cnt_tx_bit_rate <= cnt_tx_bit_rate + '1';
                end if;
        end if;
end process;
```

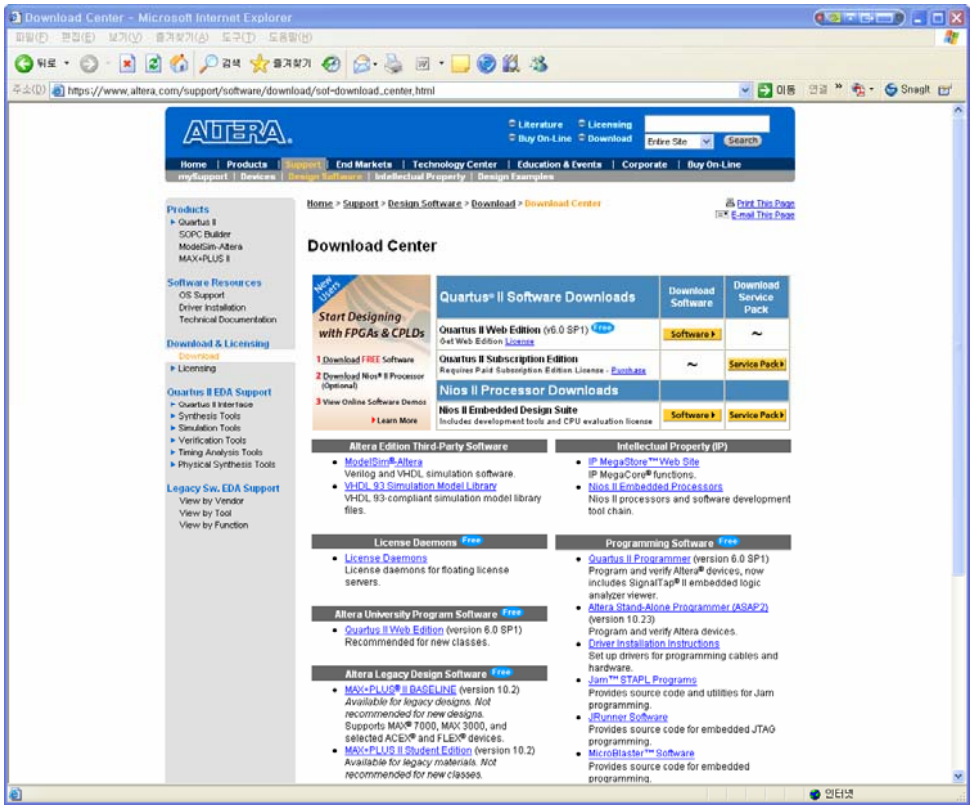
```
process (resetn, clk)
begin
	if resetn = '0' then
		cnt_tx_data <= (others => '0');
	elsif clk'event and clk = '1' then
		if tx_routine = data_tx then
			if cnt_tx_bit_rate = bit_rate then
				if cnt_tx_data = "111" then
					cnt_tx_data <= (others => '0');
				else
					cnt_tx_data <= cnt_tx_data + '1';
				end if;
			end if;
		else
			cnt_tx_data <= (others => '0');
		end if;
	end if;
end process;

process (resetn, clk)
begin
	if resetn = '0' then
		tx_data <= (others => '0');
	elsif clk'event and clk = '1' then
		if tx_routine = tx_idle then
			if txd_en = '1' then
				tx_data <= uart_data_out;
			end if;
		end if;
	end if;
end process;

process (resetn, clk)
begin
	if resetn = '0' then
		tx_parity_chk <= '0';
	elsif clk'event and clk = '1' then
		if tx_routine = tx_idle then
			if txd_en = '1' then
			tx_parity_chk <= uart_data_out(7) xor uart_data_out(6) xor
					uart_data_out(5) xor uart_data_out(4) xor
					uart_data_out(3) xor uart_data_out(2) xor
					uart_data_out(1) xor uart_data_out(0) xor
					parity_mode;
			end if;
		end if;
	end if;
end process;

process (resetn, clk)
begin
	if resetn = '0' then
		txd <= '1';
	elsif clk'event and clk = '1' then
		case tx_routine is
			when tx_idle =>
```

```
                                        txd <= '1';
                                when tx_start =>
                                        txd <= '0';
                                when data_tx =>
                                        if cnt_tx_data = "000" then
                                                txd <= tx_data(0);
                                        elsif cnt_tx_data = "001" then
                                                txd <= tx_data(1);
                                        elsif cnt_tx_data = "010" then
                                                txd <= tx_data(2);
                                        elsif cnt_tx_data = "011" then
                                                txd <= tx_data(3);
                                        elsif cnt_tx_data = "100" then
                                                txd <= tx_data(4);
                                        elsif cnt_tx_data = "101" then
                                                txd <= tx_data(5);
                                        elsif cnt_tx_data = "110" then
                                                txd <= tx_data(6);
                                        elsif cnt_tx_data = "111" then
                                                txd <= tx_data(7);
                                        else
                                                txd <= '1';
                                        end if;
                                when tx_parity =>
                                        txd <= tx_parity_chk;
                                when tx_stop =>
                                        txd <= '1';
                                when others =>
                                        txd <= '1';
                        end case;
                end if;
        end process;

        end hb;
```

7.6 Altera 설계 소프트웨어 설치 방법(Quartus II)

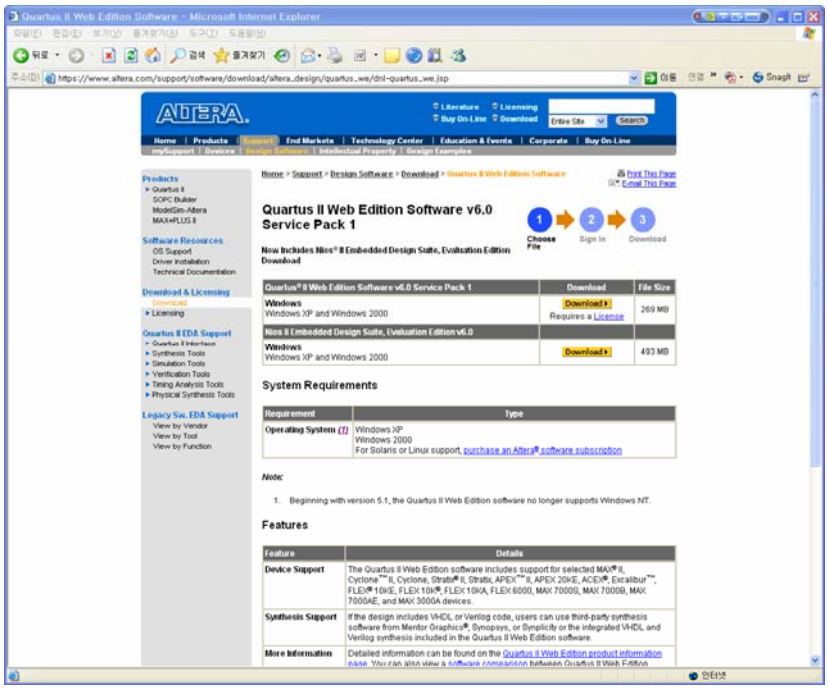
- Altera에서 제공하는 설계 소프트웨어인 Quartus II를 설치하기 위해서는 Altera 홈페이지(http://www.altera.com/)의 다운로드 센터에서 받을 수 있습니다.
- 여기에서는 설계 소프트웨어인 Quartus II 뿐만 아니라, 디바이스의 핀 정보, 핀 맵, 디바이스 관련 회로 등의 모든 정보를 여기에서 찾아서 확인할 수 있습니다. 다음의 그림에서는 Altera 홈페이지를 보여주고 있습니다.

- 홈페이지의 상단에 Download란 버튼을 클릭하면 다운로드 센터로 들어가게 됩니다. 다운로드 센터에는 설계 소프트웨어인 Quartus II Web Edition을 받을 수 있고, 과거의 설계 소프트웨어인 Max+plus II 또한 같은 방법으로 다운 받아서 사용할 수 있습니다.
- 또한 여기에서 시뮬레이션을 할 수 있는 소프트웨어인 ModelSim도 지원해 주고 있어서, Quartus II에서의 기본으로 지원하는 시뮬레이션 과정에서 벗어나 좀 더 정밀한 시뮬레이션 검증을 할 수 있도록 지원해 주고 있습니다.
- 다음 그림에 Download Center의 화면을 보여 주고 있습니다. 현재 그림에서는 Quartus II Web Edition V6.0을 지원하고 있는 모습을 보여주고 있습니다. 이러한 소프트웨어의 버전은 시간에 따라 계속 업 그레이드 하고 있습니다.

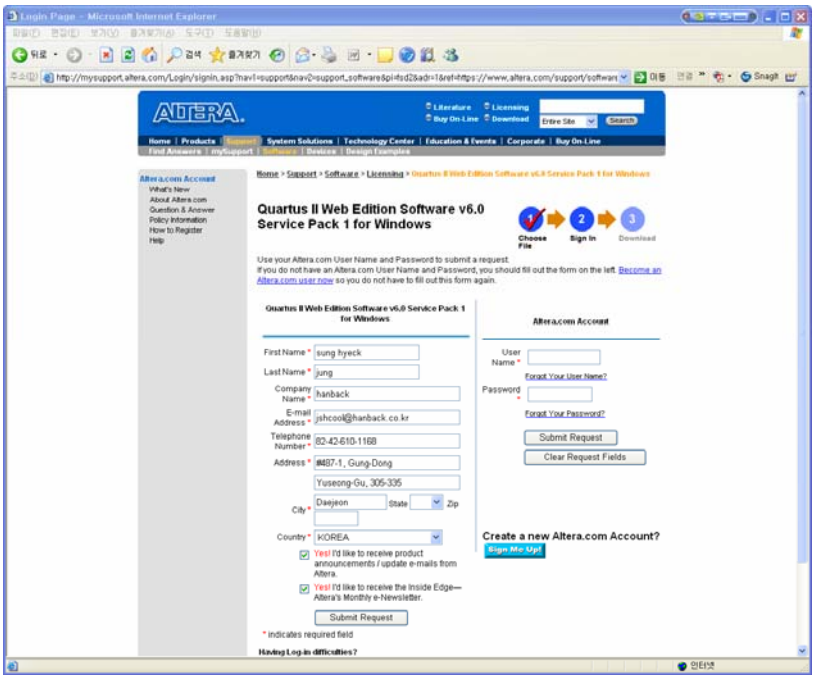


❑ 다운로드 센터에서 표에 있는 Software를 클릭해서 넘어가면 Quartus II Web Edition을 다운 받는 과정을 진행하게 됩니다.

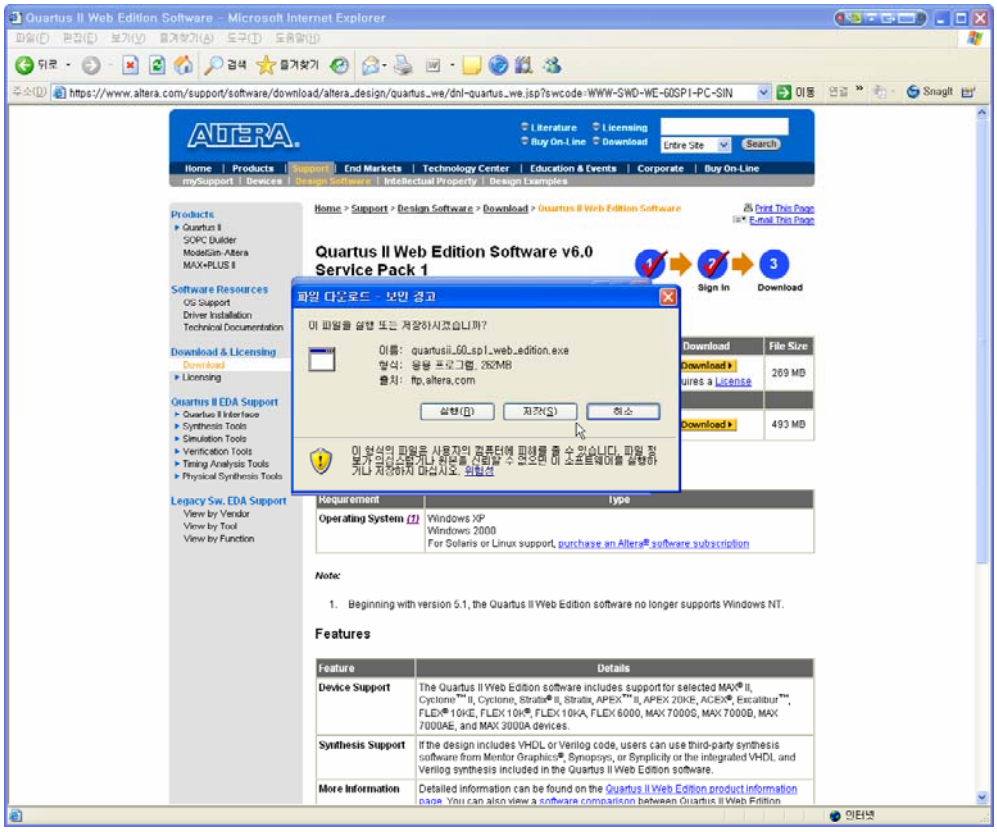
❑ 처음 과정은 Choose File로 Quartus II와 Nios II의 어떠한 소프트웨어를 다운 받을 지 결정하는 단계 입니다. 따라서 소프트웨어 선택하는 부분과 관련 소프트웨어의 지원 디바이스 및 정보 등이 나와 있습니다. 여기에서는 Quartus II에 있는 Download 버튼을 클릭하여 다음으로 넘어가면 되겠습니다.



❑ 다음으로는 Sign in 과정으로 다운 받으려는 사람의 기본 정보를 입력하는 과정입니다. 가입이 되어 있는 사람은 User Name와 Password만 가지고 이 과정을 넘어갈 수 있고, 아니면 화면의 왼쪽 그림과 같이 기본 정보를 입력하면 됩니다.

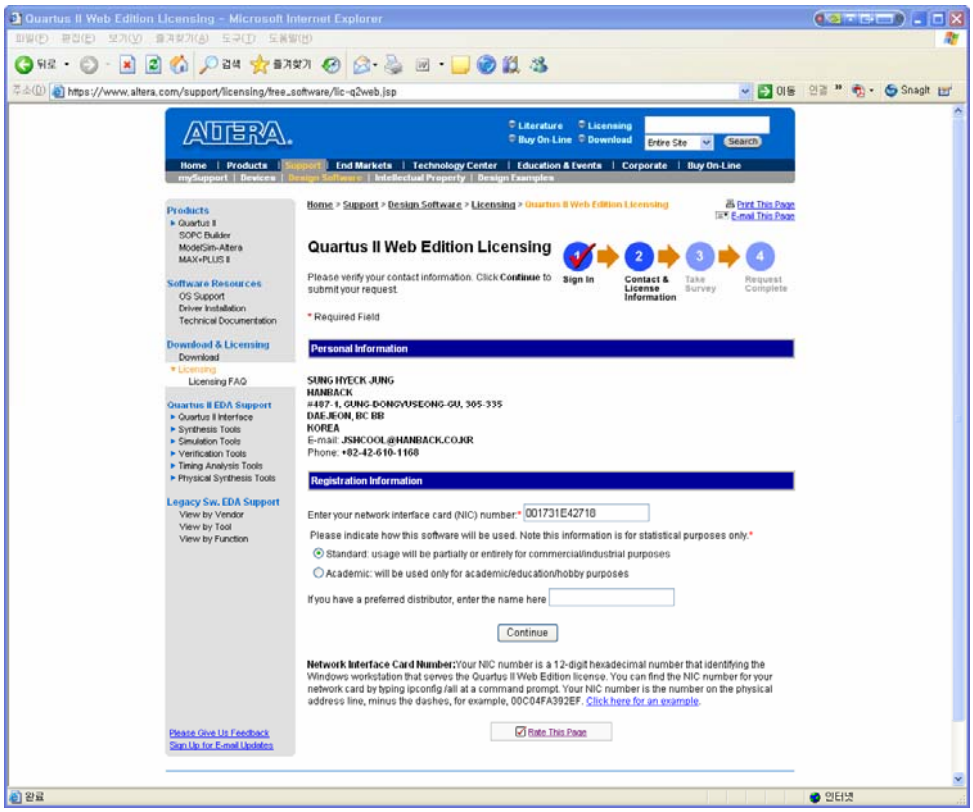


❑ 이 과정을 다 마치면 다음 과정에 파일을 받게 됩니다. 여기에서 저장을 눌러 폴더를 지정하여 다운 받으면 됩니다.



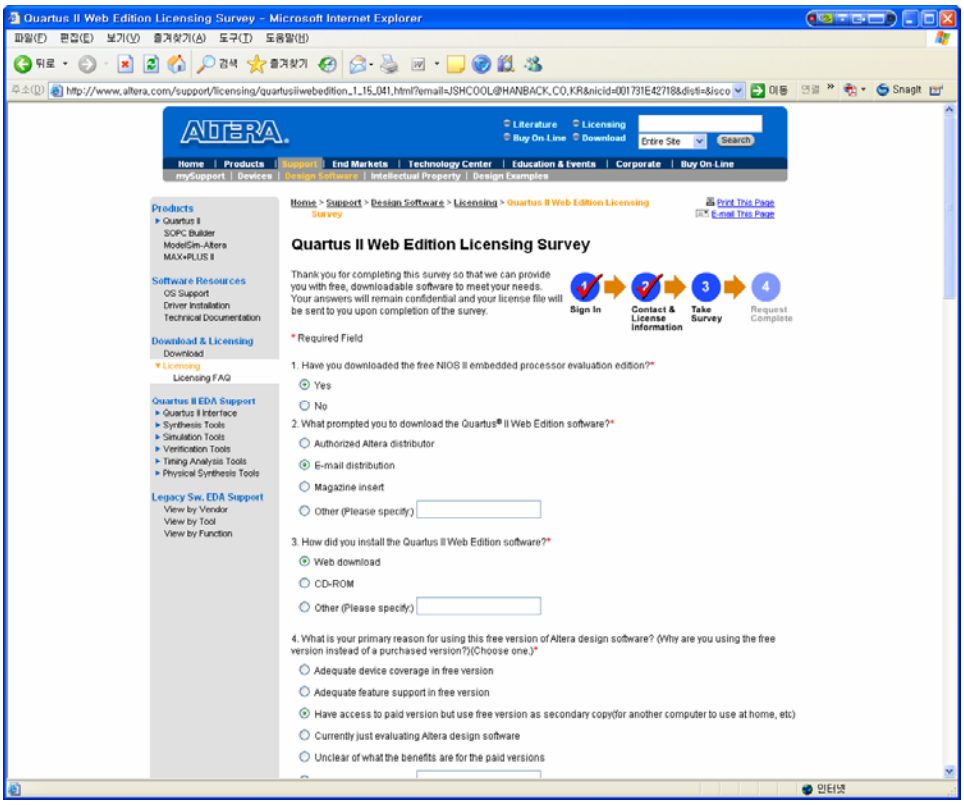
❑ Quartus II의 작업의 다운이 완료되면 Quartus II를 구동하기 위한 라이센스 파일을 다운 받아야 합니다. 따라서 설계 소프트웨어와 라이센스의 총 2개를 파일을 다운받아야 합니다. 여기에서는 이전의 작업에서 Quartus II의 다운로드 버튼 및 에 있는 Requires a License의 License 버튼을 클릭하여 라이센스 다운 작업을 진행할 수 있습니다. 현재 창은 Contact & License Information 작업 과정으로 이전의 작업에서 기본 정보를 입력을 하고 계속 진행하는 상태라 따로 정보를 입력할 필요가 없습니다.

❑ 다음의 그림에서 사전에 입력 된 기본 정보에 대한 사항을 확인할 수 있습니다. 라이센스 파일을 받기 위해서는 설치하려는 컴퓨터의 Network interface card(NIC) number의 12자리를 입력해 주어야 합니다. 이것은 도스 창의 명령 프로포트에서 ipconfig /all 이라고 입력하면 프로그램을 설치할 컴퓨터의 Physical Address를 보여줍니다. 따라서 이 12자리의 값을 웹 브라우저에 입력해 주면 됩니다. 또한 다음에 묻는 소프트웨어 사용상의 질문을 체크해 주고 다음 과정으로 넘어가면 됩니다. 다음의 그림에 Contact & License Information 작업의 그림이 나와 있습니다.



❑ 다음의 진행 사항은 Take Survey로서 현재 다운 받으려는 사람이 Altera 회사의 얼마나 이용하고 있는지를 묻는 것 입니다.

❑ 따라서 사용자는 질문을 보고 해당사한에 체크를 하고 다음으로 진행해 주면 됩니다. 다음의 그림에는 해당되는 질문에 체크하는 과정을 보여주고 있습니다.



❑ 이전까지의 작업으로 라이센스의 다운 과정을 모두 마무리 한 것입니다. 이 라이센스 파일은 사전에 등록한 메일 주소에 파일 형태로 첨부되어 받게 됩니다. 또한 확장자는 .dat 파일로 라이센스 진행 과정에서 등록한 Network interface card(NIC) number를 가지고 생성한 라이센스 파일이기 때문에 다른 컴퓨터에서 사용이 불가능 합니다.

❑ 만약 다른 컴퓨터에서 사용하기 위해서는 위의 다운로드 센터에서 라이센스 다운의 작업을 따로 해 주어야 합니다. 라이센스만 다운 받는 작업을 할 때는 Quartus II 소프트웨어를 받을 때의 기본 정보 입력 과정이 추가됩니다. 다음의 그림에서 라이센스에 대한 다운 작업이 완료된 그림을 보여주고 있습니다. 또한 메일로 발송했다는 정보와 메일이 지연될 수 있다는 내용을 보여주고 있습니다.

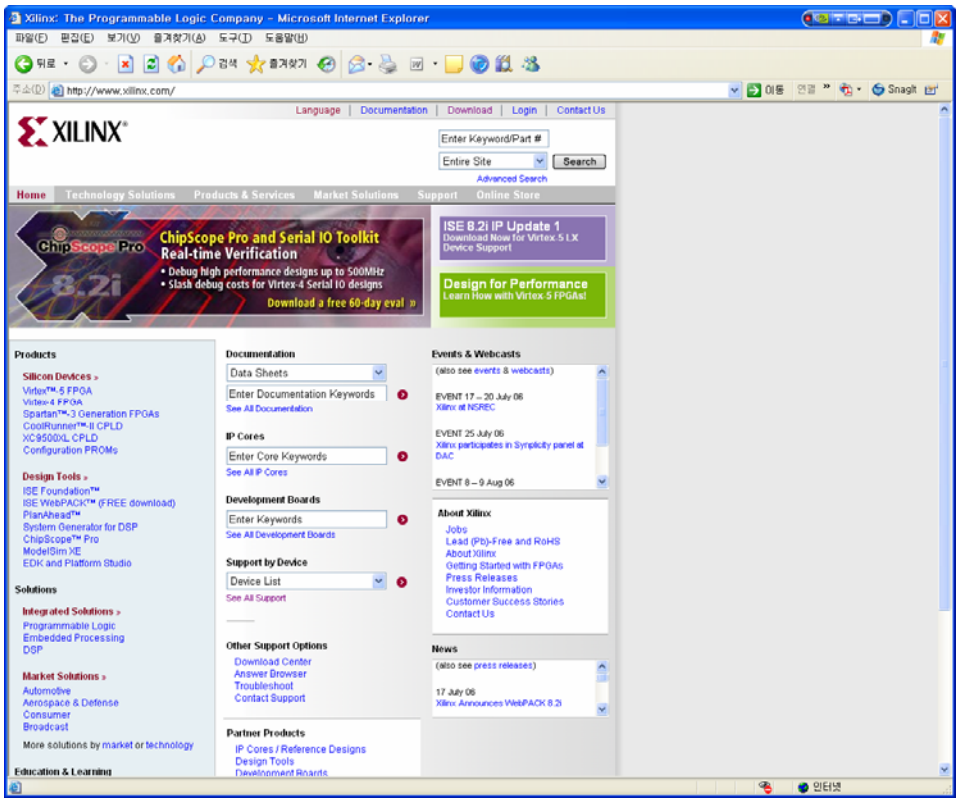
❑ 이상의 다운 작업을 마쳤으면 소프트웨어를 컴퓨터에 설치하고 다운 받은 라이센스 파일을 Quartus II를 실행하여 등록해 주면 됩니다.

❑ 라인센스 파일은 웹에서 무료로 배포하고 있는 파일로 사용 기간 제한이 있습니다. 따라서 이 기간이 지났을 때 라이센스 다운 작업을 다시 하셔서 등록해 주면 됩니다.

7.7 Xilinx 설계 소프트웨어 설치 방법(ISE)

❑ Xilinx에서 제공하는 설계 소프트웨어인 ISE를 설치하기 위해서는 Xilinx 홈페이지(http://www.xilinx.com/)의 다운로드 센터에서 받을 수 있습니다.

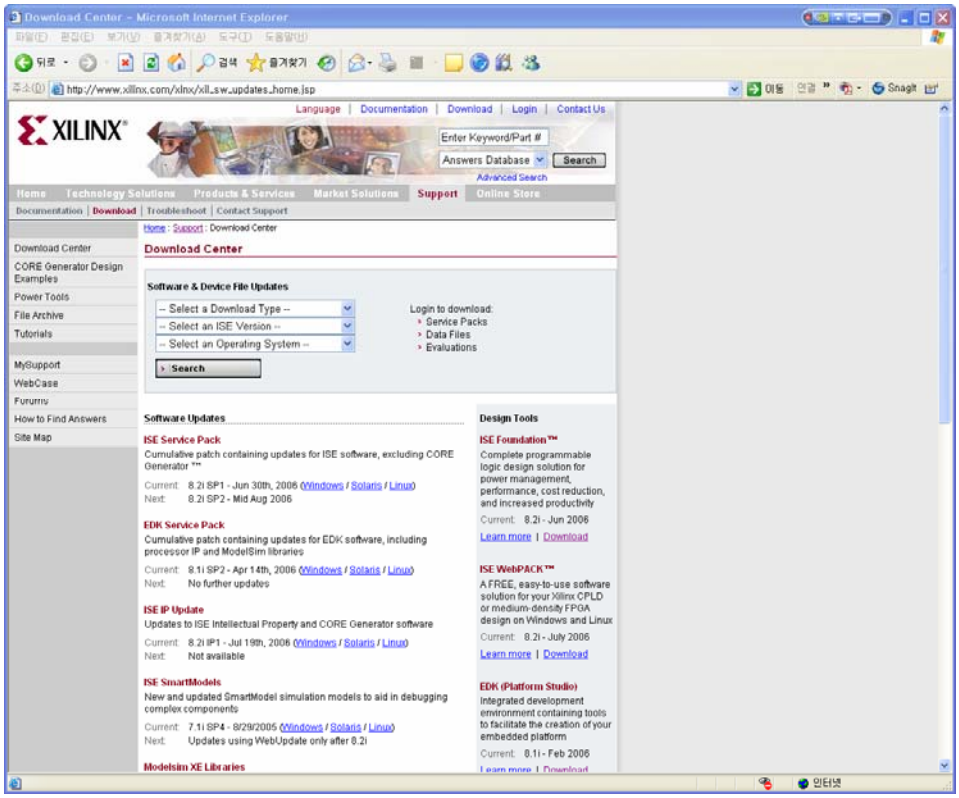
❑ 여기에서도 설계 소프트웨어인 ISE 뿐만 아니라, 디바이스의 핀 정보, 핀 맵, 디바이스 관련 회로 등의 모든 정보를 여기에서 찾아서 확인할 수 있습니다. 다음의 그림에서는 Xilinx 홈페이지를 보여주고 있습니다.



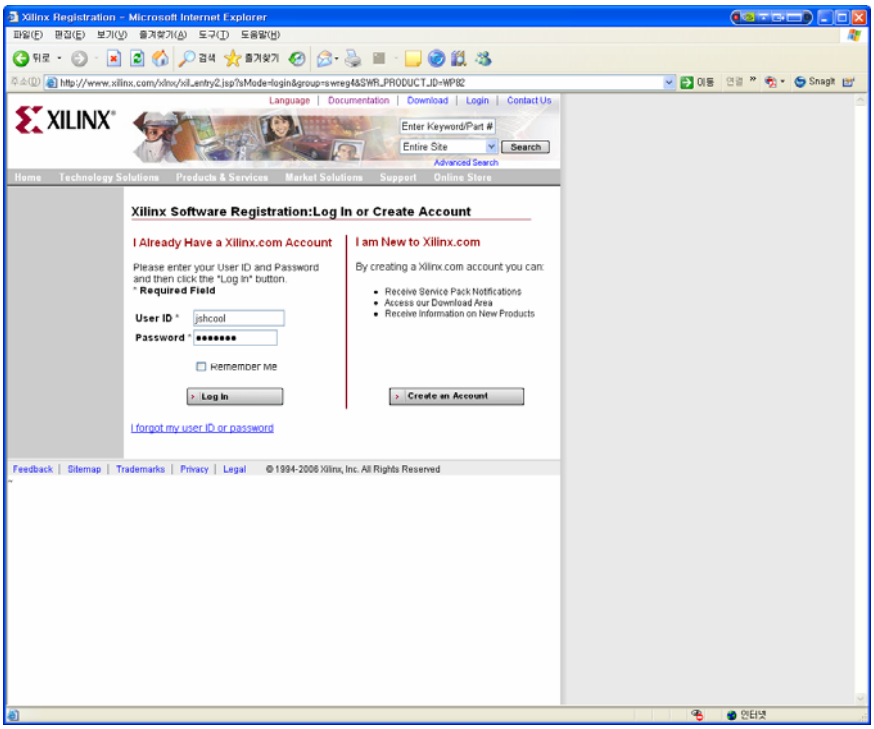
❑ ISE도 Xilinx의 다운로드 센터에서 소프트웨어를 다운 받아 사용이 가능합니다. 홈페이지의 상단의 메뉴에서 Download를 클릭하여 다운로드 센터로 들어갑니다. 다운로드 센터에는 Software Updates와 Design Tools이 있습니다.

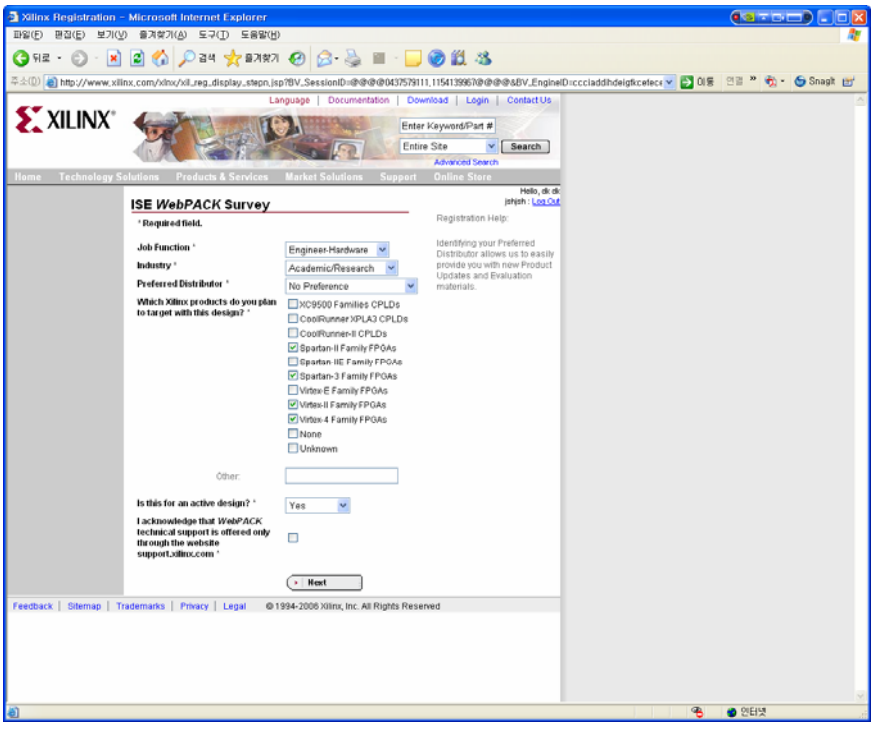
❑ 따라서 Xilinx 디바이스와 관련 된 모든 설계 소프트웨어를 다운 받아서 사용이 가능합니다. 다운로드 센터에는 Design Tool에 대한 업데이트 파일과 디바이스를 지원하기 ModelSim등의 프로그램이 있습니다.

❑ 여기에서는 Design Tools에서 ISE WebPACK을 선택하여 다운 받으면 됩니다. 다음의 그림에서는 다운로드 센터의 설계 검증 프로그램 및 업데이트 파일에 대한 목록을 보여주고 있습니다.



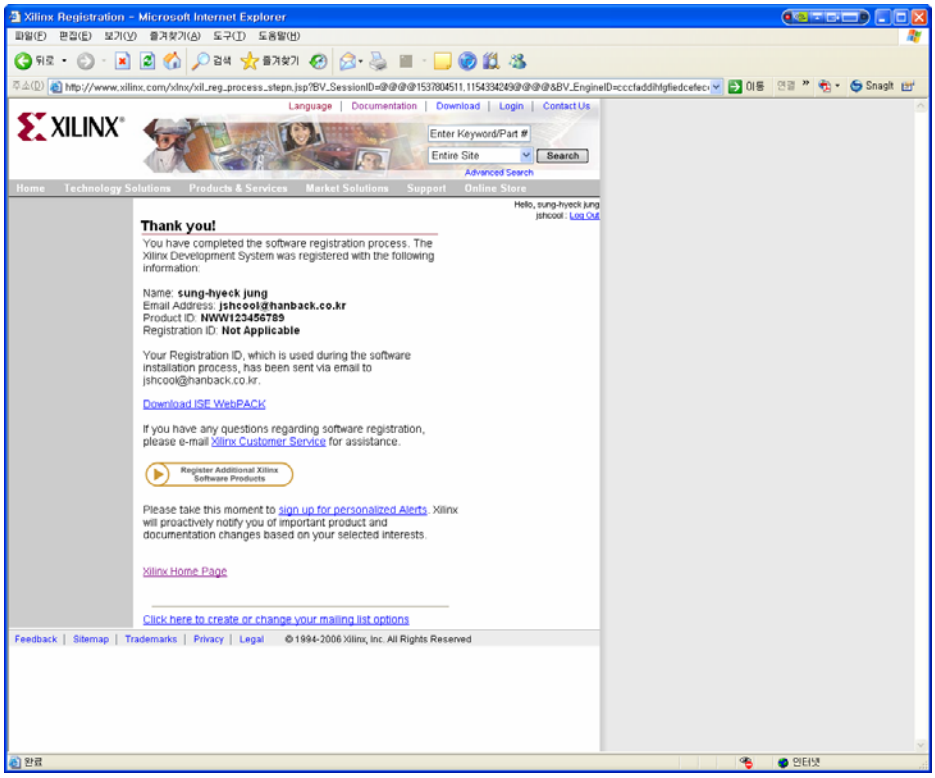
- ISE에 다운 버튼을 클릭하면 Xilinx 설계 소프트웨어를 다운 받는 작업을 진행하게 됩니다.
- 다음 단계에는 사전에 등록된 아이디가 있으면 아이디를 입력하고 다음 단계로 넘어가고, 아니면 오른 쪽의 Create an Account를 클릭하여 사용자의 기본 정보를 입력하게 됩니다. 초기에는 다운에 관한 기본 정보 입력을 보여주고 있습니다. 따라서 사용 받으려는 사람의 정보를 선택해 주면 됩니다. 다음 그림은 해당 사항을 선택한 모습을 보여주고 있습니다. 왼쪽의 그림은 사전에 등록된 아이디를 가지고 로그인하는 모습을 보이고 있고, 오른 쪽 그림은 다운 받는 사람에 대한 추가 정보를 입력하는 창이 되겠습니다.



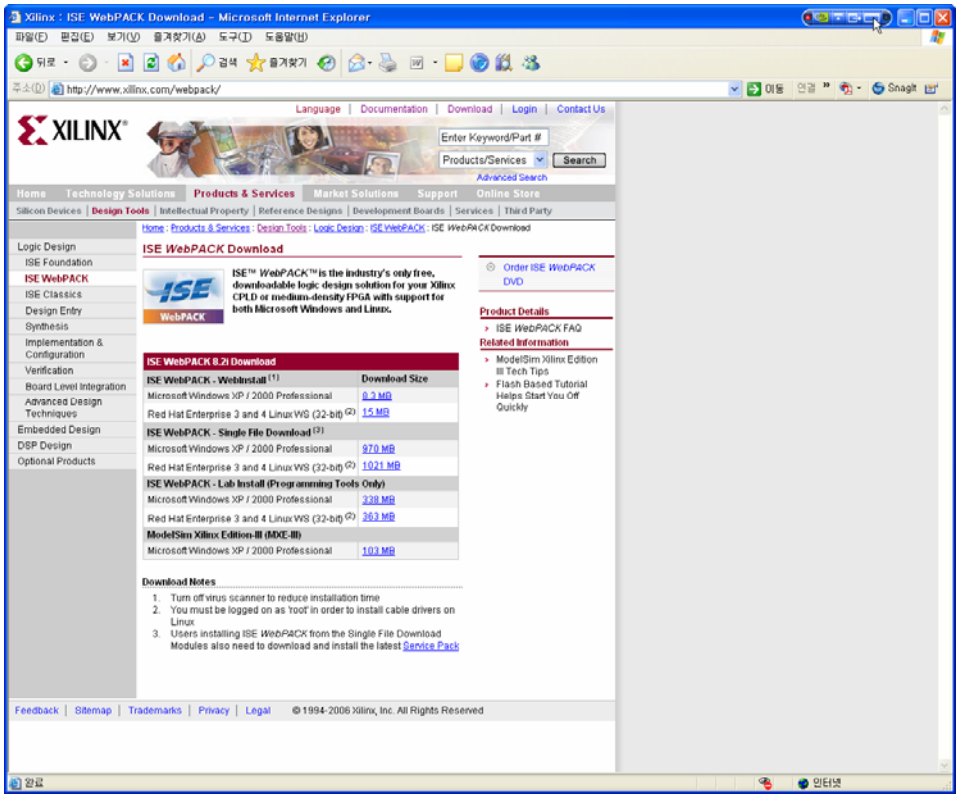


❑ 이전까지 작업으로 Xilinx ISE WebPACK를 받을 준비는 다 되었습니다. 다음의 그림에서는 정보

입력에 대한 사항을 마지막으로 보여 주고 있습니다. 창에서 중간에 있는 Download ISE WebPACK를 클릭하여 다음 단계로 넘어갑니다.



- 이 창에서는 단운로드 단계의 마지막 단계로서 ISE WebPACK를 다운 받을 수 있습니다. 이 창에서 저장하려는 파일의 사이즈를 클릭하여 프로그램을 다운할 수 있습니다.
- 이렇게 파일을 다운받고 저장이 끝나면 컴퓨터에 프로그램을 설치해 주면 됩니다. 여기는 Quartus II와는 달리 별도의 라이센스 파일을 다운할 필요는 없습니다. 다음 그림은 다운 로드의 마지막 단계를 보여주고 있습니다.


Xilinx : ISE WebPACK Download - Microsoft Internet Explorer
http://www.xilinx.com/webpack/
XILINX
Language | Documentation | Download | Login | Contact Us
Enter Keyword/Part #
Products/Services
Search
Advanced Search
Home
Technology Solutions
Products & Services
Market Solutions
Support
Online Store
Silicon Devices | Design Tools | Intellectual Property | Reference Designs | Development Boards | Services | Third Party
Home : Products & Services : Design Tools : Logic Design : ISE WebPACK : ISE WebPACK Download
Logic Design
ISE Foundation
ISE WebPACK
ISE Classics
Design Entry
Synthesis
Implementation & Configuration
Verification
Board Level Integration
Advanced Design Techniques
Embedded Design
DSP Design
Optional Products
ISE WebPACK Download
ISE™ WebPACK™ is the industry's only free, downloadable logic design solution for your Xilinx CPLD or medium-density FPGA with support for both Microsoft Windows and Linux.
Order ISE WebPACK DVD
Product Details
ISE WebPACK FAQ
Related Information
ModelSim Xilinx Edition III Tech Tips
Flash Based Tutorial Helps Start You Off Quickly
ISE WebPACK 8.2i Download
ISE WebPACK - WebInstall (1)
Download Size
Microsoft Windows XP / 2000 Professional
8.2 MB
Red Hat Enterprise 3 and 4 Linux WS (32-bit) (2)
15 MB
ISE WebPACK - Single File Download (3)
Microsoft Windows XP / 2000 Professional
970 MB
Red Hat Enterprise 3 and 4 Linux WS (32-bit) (2)
1021 MB
ISE WebPACK - Lab Install (Programming Tools Only)
Microsoft Windows XP / 2000 Professional
338 MB
Red Hat Enterprise 3 and 4 Linux WS (32-bit) (2)
363 MB
ModelSim Xilinx Edition-III (MXE-III)
Microsoft Windows XP / 2000 Professional
103 MB
Download Notes
1. Turn off virus scanner to reduce installation time
2. You must be logged on as 'root' in order to install cable drivers on Linux
3. Users installing ISE WebPACK from the Single File Download Modules also need to download and install the latest Service Pack
Feedback | Sitemap | Trademarks | Privacy | Legal
© 1994-2006 Xilinx, Inc. All Rights Reserved