“내 칩 제작 서비스” 오픈-소스 디자인 킷/연구노트 13:

Arduino, Vitis-HLS, Quartus

연구과제명	반도체 기술 개발 지원 고경력 전문인력 활용 사업(25JB1710)
연구기간	2025 년 6 월~2026 년 12 월
연구책임자	고상춘
기록자	국일호
확인자	
작성일자	2025 년 8 월 30 일

ETRI 한국전자통신연구원
www.etri.re.kr

"내 칩 제작 서비스" 오픈-소스 디자인 킷/연구노트13

Arduino, Vitis-HLS, Quartus

목차:

6. 맺음말

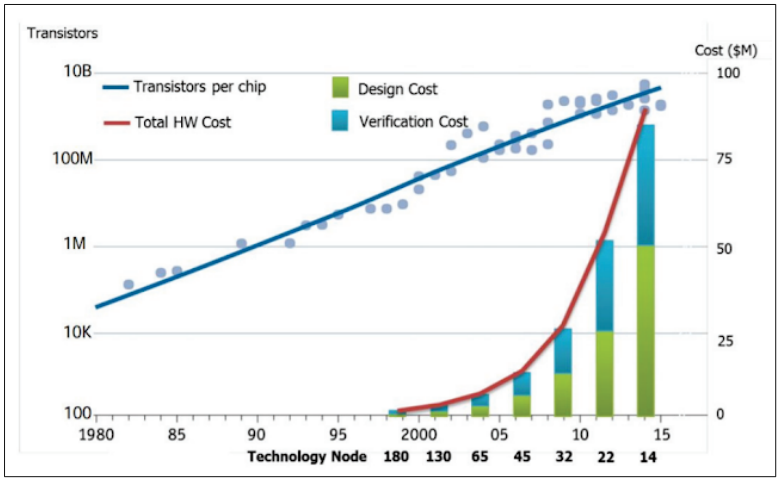
by GoodKook, goodkook@gmail.com

1. 개요

우리는 내 칩을 설계 하기 위해 남이 만든 칩을 활용 한다. 남이 만든 칩으로 꾸민 컴퓨터에서 컴퓨팅 언어로 알고리즘을 기술하고 도면을 그리며 시뮬레이션을 수행한다. "내 칩"을 만들기 위해 남이 만들어 놓은 컴퓨터를 활용하려면 하드웨어는 물론 소프트웨어에 이르기까지 폭 넓은 이해가 있어야 한다. 게다가 "내 칩"을 제작하려면 상당한 비용이 드는데, 다행히 "내 칩 제작 서비스" 사업으로 무료 제작해 준다[내 칩 제작 서비스]. 공고문에 따르면 도면(레이아웃)을 제출 하고 내 칩을 받기까지 무려 반년의 기간이 소요된다. 시간을 감안 하면 공짜가 아니라는 뜻이다. 수개월 후에 받아 든 "내 칩"이 내가 설계한 의도대로 작동하길 바란다면 확실히 검증되어야 한다. 시뮬레이터는 반도체 설계의 검증용으로 만들어 진 응용 프로그램이다. 이 복잡한 응용 프로그램에 오류는 없을까? 내가 만든 테스트 벤치에 예기치 못한 헛점은 없을까?

남이 만들어 놓은 칩 중에 특별히 기능을 정해 놓지 않은 것들로 마이크로 프로세서와 FPGA가 있다. 마이크로 프로세서에 프로그램을 써 넣어 마치 "내 칩" 처럼 작동 하도록 흉내 낼 수 있다. "내 칩"에서 작동할 알고리즘의 검증을 비록 남의 칩이지만 하드웨어로 검증이 가능 하다. 그런데 "내 칩"은 RTL 이지만 "남의 칩"에 구현된 알고리즘은 "시스템 수준" 알고리즘이다. "내 칩"의 최종 추상화 목표와 마이크로 프로세서라는 "남의 칩"에 구현된 알고리즘 사이에는 추상성 격차가 있다. 이런 추상성 격차가 있음에도 불구하고 굳이 남의 칩, 마이크로 프로세서를 동원하여 검증할 필요가 있을까? "내 칩"의 검증을 위해 작성한 테스트 벤치에 물림으로써 시뮬레이션 모델에 대한 걱정을 덜 수 있다.

FPGA는 RTL에서 "내 칩"을 흉내 낼 수 있도록 재구성이 가능한 "남의 칩"이다. 앞으로 만들어 질 "내 칩"에 버금가는 낮은 추상화 수준, 즉 RTL에서 검증이 가능하다. 실제 하드웨어를 구축해야 하는 만큼 시험 환경을 갖추는데 소요 시간과 비용이 매우 높다. 낮은 추상화 수준의 하드웨어에서 오류를 찾고 수정하려면 매우 큰 수고가 든다. 반도체 설계용 응용 프로그램이 충분히 성숙되어 시뮬레이션으로도 충분하다는 지금 굳이 많은 비용을 들여가며 FPGA를 활용하여 "내 칩"을 검증 할 필요가 있을까? RTL 시뮬레이션 응용 프로그램들은 설계자의 구문을 해석하여 마치 하드웨어가 있는 것처럼 흉내 낸다. 설계(행위를 묘사한 구문)로 부터 하드웨어로 변환하는 도구는 따로 있다. 시뮬레이터와 합성기는 받아들일 수 있는 구문의 추상성 수준의 차이가 존재한다. 뿐 만 아니라 코딩 스타일[연구노트3]에 따라 합성기는 시뮬레이터의 동작과 전혀 다른 회로를 만들어 낸다. 설계 초기 단계에서 간과된 시뮬레이터와 합성기의 작은 불일치가 시스템 설계를 망칠 수 있다. 반도체 시장 관련 보고서(논문 등)에서 설계 규모의 증가에 비해 검증 비용의 급격한 증가를 가장 큰 위협이라고 보고하는 이유다.



[출처] OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain

"남의 칩"을 활용하면서 얻는 또 다른 이점으로 검증 시간의 단축을 들 수 있다. 시제품을 만들어보는 의미로 프로토 타이핑(proto-typing)이 있다. 반도체 설계에서 프로토-타이핑은 "내 칩"의 설계에 "남의 칩(특히 FPGA)"을 적극적으로 활용하는 방법론이다. 반도체 설계에서 프로토 타이핑에 들이는 비용은 상당하다. 반도체 칩은 시제품 제작에 엄청난 비용이 들 뿐만 아니라 일단 칩이 만들어 진 후 실수가 발견되면 재 제작 외에 다른 방법은 없다. 오류를 찾아 수정할 수 있는 초기 개발 NRE(Non-Recursive Engineering)비용이 매우 높기 때문에 검증의 비중이 높아질 수 밖에 없다.

프로그래밍 언어 C/C++를 이해하고 있다면 "내 칩"을 만들 수 있다. 이 문서는 이에 필요한 반도체 설계 자동화 도구들을 소개한다. 예제는 C 로 작성한 디지털 FIR 필터다[연구노트9]. 최고의 수준에서 가장 낮은 회로 단계의 레이아웃에 이르기까지 급격한 추상성 변화의 과정에 자동화 도구들이 활용 된다. 자동화 도구를 무작정 신뢰하기에는 위험성이 크다. 오픈-소스 도구를 사용하는 경우 신뢰도는 설계자 스스로 담보해야 한다. 어떤 도구를 사용하든 자동화 도구들이 내놓은 결과물의 검증은 설계자의 몫이다. 성공적인 "내 칩" 만들기에 "남의 칩"과 도구들을 적극적으로 활용하는 방법을 알아보기로 한다.

2. Arduino

아듀이노는 Atmel 사의 8-비트 또는 32-비트 마이크로 컨트롤러를 장착한 보드 응용 보드로 시작 됐다. 오픈-소스 하드웨어로서 회로도는 물론 개발 도구가 모두 공개되어있다. 이에 수많은 사용자들이 참여하여 범용 내장형 시스템의 개발 플랫폼이 되었다. GNU C/C++ 컴파일러가 준비되었을 경우 Atmel 뿐만 아니라 라즈베리 파이(Raspberry Pi), ESP32 등 다양한 마이크로 컨트롤러를 수용한다. 이루 헤아릴 수 없을 만큼 다양한 응용 보드들이 구동 라이브러리와 함께 공개적으로 제공된다. 교육용은 물론 전문 연구 개발의 영역에서도 널리 활용되고있다.

2-1. Arduino 설치

a. Arduino IDE(Integrated Development Environment)

Arduino-IDE는 아듀이노 보드에 응용 프로그램을 작성하고 컴파일된 바이너리 코드를 올려 놓을 수 있는 통합 개발환경이다. 이 개발 환경은 윈도우즈와 리눅스 운영체제에서 작동한다. "내 칩 설계 환경[연구노트2]"의 일부로 리눅스용 Arduino-IDE의 설치하는 방법을 설명한다. 아래 다운로드 링크 페이지에서 "Linux AppImage (64-bit X86-64)" 를 내려받는다. 편의상 도구들을 디렉토리 /opt/Arduino 에 저장해 둔다.

https://www.arduino.cc/en/software/

또는

```
$ sudo mkdir /opt/Arduino
$ cd /opt/Arduino
$ sudo wget https://downloads.arduino.cc/arduino-ide/arduino-ide_2.3.6_Linux_64bit.AppImage
```

내려받은 파일 명은 arduino-ide_2.3.6_Linux_64bit.AppImage 다. 이 파일이 실행 속성을 가지지 않고 있다면 이를 변경한다.

```
$ sudo chmod +x arduino-ide_2.3.6_Linux_64bit.AppImage
```

AppImage 파일은 소스 코드 편집기, 타깃 마이크로 컨트롤러의 C/C++ 컴파일러 운용 그리고 바이너리 이미지 올리기(upload)등 일련의 개발 도구들을 패키지화 하여 한개의 파일에 모아놓은 것이다. AppImage를 실행시키려면 libfuse2 가 설치되어야 한다.

```
$ sudo apt install libfuse2
```

[주] libfuse2 는 "내 칩 디자인 킷"의 도구와 함께 이미 설치했다[연구노트2].

임의로 내려받아 저장한 응용 프로그램을 리눅스의 파일 시스템 실행 경로에 포함 시키기 위해 아래와 같이 소프트 링크(바로가기)를 만들어 준다.

```
$ sudo ln -s /opt/Arduino/arduino-ide_2.3.6_Linux_64bit.AppImage /usr/bin/arduino-ide
```

어느 디렉토리 위치 에서도 아듀이노 개발 환경을 사용할 수 있다.

[주] arduino-ide가 샌드박스(SandBox) 보안 때문에 실행 되지 않을 경우 --no-sandbox 스위치를 주고 실행한다.

b. Arduino CLI (Command-Line Interface)

Arduino IDE는 그래픽 사용자 인터페이스 환경(GUI, Graphic User Interface)이다. 프로그램 개발에 필요한 모든 도구들을 통합한 환경은 사용이 편리 하지만 개발 중 컴파일과 바이너리를 타깃 보드에 올리기 위해 반복적으로 GUI 를 띄우려면 번거롭다. 매번 여러 단계 메뉴를 선택하고 버튼을 눌러줘야 한다. 번거롭게 느끼는 틈으로 사용자의 실수가 끼어든다. 아듀이노는 GUI외에 명령줄 환경을 제공한다. Makefile 스크립트를 활용할 수 있다. 별도의 설치 방법은 없다. 아래처럼 링크에서 내려 받아,

```
$ cd /opt/Arduino
$ sudo wget https://downloads.arduino.cc/arduino-cli/arduino-cli_latest_Linux_64bit.tar.gz
```

압축을 풀어놓고,

```
$ sudo tar xvf arduino-cli_latest_Linux_64bit.tar.gz
```

바로가기 링크를 건다.

```
$ sudo ln -s /opt/Arduino/arduino-cli /usr/bin/arduino-cli
```

2-2. 아듀이노 타깃 보드

아듀이노는 마이크로 컨트롤러 보드 회로도와 개발 도구를 모두 공개하였다. 타깃 마이크로 컨트롤러로 Atmel 사의 8-비트 또는 32-비트 마이크로 컨트롤러를 장착한 보드를 출시하였다. 오픈-소스 하드웨어 개발 환경이 모두 공개된 탓에 각종 응용 라이브러리는 물론 다양한 응용 보드들이 많은 개발자들에 의해 자발적으로 추가되었다. Atmel 사 뿐만 아니라 라즈베리 파이(Raspberry PI), ESP32, RISC-V 등 다수의 마이크로 컨트롤러(GNU C/C++ 컴파일러가

준비된)를 채택한 보드들도 아듀이노 개발 환경에서 사용 할 수 있게 되었다. 지금의 아듀이노는 마이크로 컨트롤러 특정하지 않고 범용 개발 환경으로서 더 큰 가치를 가진다. 오픈-소스 하드웨어 집단 지성의 표본이라고 하겠다.

"내 칩 디자인 킷[링크]"의 검증과 칩 테스트에 아듀이노 개발환경을 사용할 것이다. 아듀이노 보드로 Atmel의 ARM 코어를 내장한 Arduimo DUE, Raspberry Pi Pico 그리고 Xtensa/RISC-V 코어가 내장된 ESP32-S/C 보드를 사용한다. 이들 보드들은 매우 저렴한 가격으로 오픈-마켓에서 판매되고 있다.

a. Arduino DUE

Arduino DUE는 32비트 ARM 코어를 내장한 Atmel 사의 마이크로 컨트롤러, SAM3X8E 를 장착한 개발 보드다. 아래 링크에서 이 보드의 기술 정보를 찾을 수 있다.

https://docs.arduino.cc/hardware/due/

[주] 오픈-마켓에서 구입 할 수 있는 Arduino DUE 보드

아뉴이노는 마이크로 컨트롤러 내장형 개발 플랫 폼이다. 특정 마이크로 컨트롤러용 개발 도구들을 따로 설치해 주어야 한다. 타깃 보드 Arduino DUE 용 개발 도구(C/C++ 컴파일러와 업로더)를 설치하는 절차는 아래와 같다.

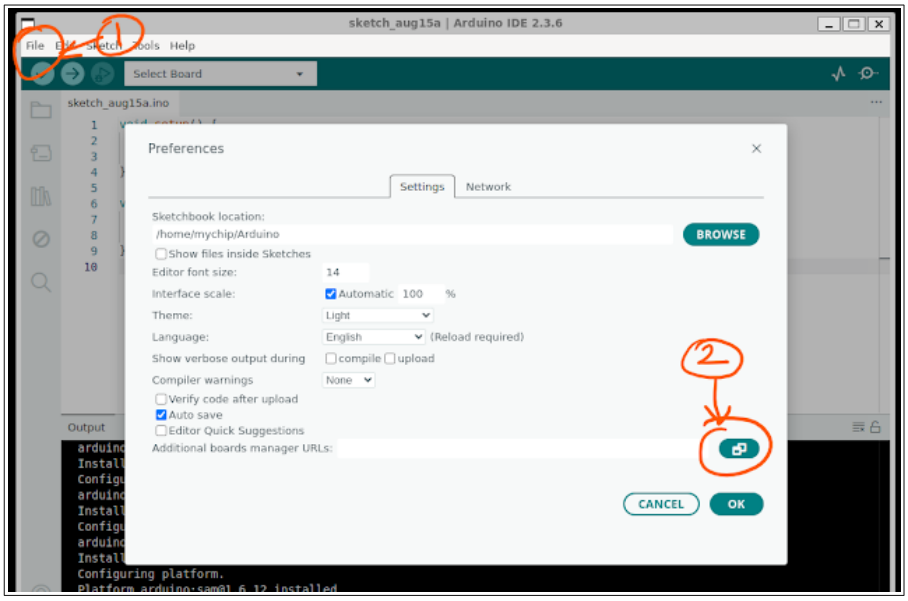
(1) 보드 매니져

(2) "DUE" 보드 검색

(3) 설치

b. Raspberry Pi Pico

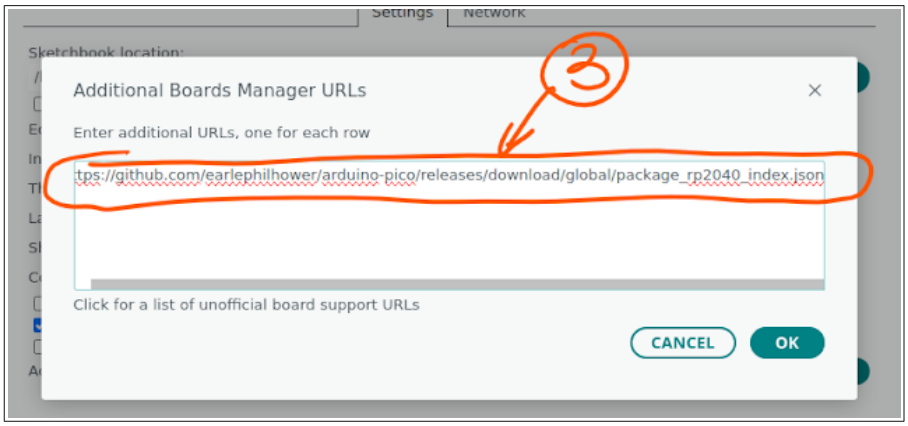
"라즈베리 Pi Pico/RP2040" 보드는 아듀이노의 공식 보드 매니져에 포함되어 있지 않다. 다음 절차에 따라 깃-허브에서 내려 받는다.



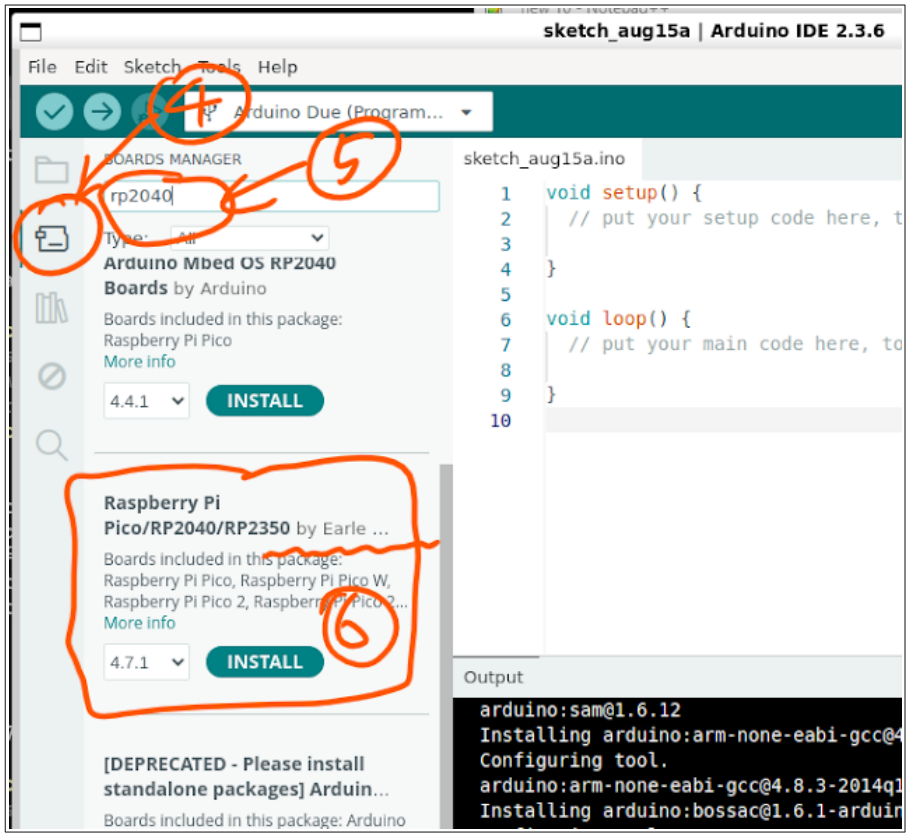
(1) 메뉴 File -> Preference

(2) 보드의 개발도구 URL 추가

https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json



(3) 깃-허브 링크 추가 후 OK



(4) 보드 매니져

(5) RP2040 검색

(6) Earlephilhower의 RP2040/2050 선택 설치

[주] 오픈-마켓에서 구입할 수 있는 Raspberry Pi Pico 보드.

c. ESP32

에스프레시프 시스템즈(Espressif Systems) ESP32 보드의 개발 환경 설치해보자. ESP32 를 채택한 보드가 다수 발매되고 있다. 아래의 예는 에스프레시프 시스템즈사에서 발매한 보드를 사용하는 경우다.

(1) 보드 매니저

(2) ESP32 검색

(3) Espressif System의 esp32 선택 설치

[주] 오픈-마켓에서 구입할 수 있는 ESP32 보드,

2-3. 아듀이노 타깃 보드의 리눅스 연결

a. 타깃 보드(윈도우즈 USB 장치)를 WSL로 연결

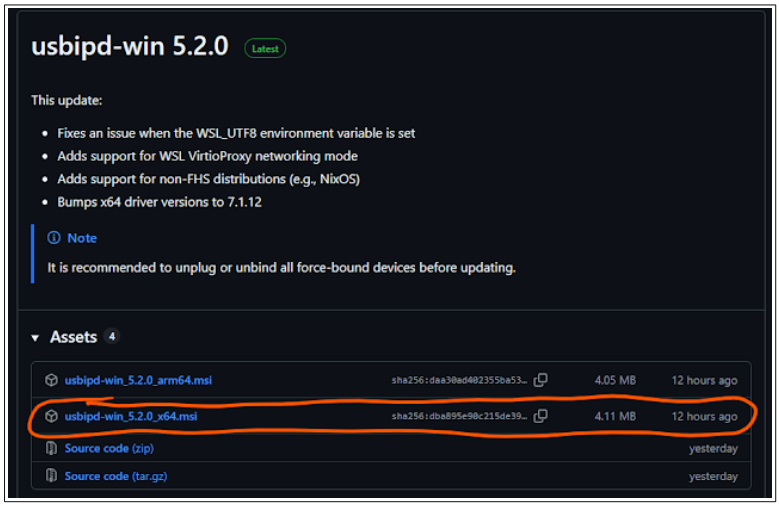
WSL은 USB 장치를 지원하지 않기 때문에 직접 연결 할 수 없다. 아듀이노 개발 환경에서 타깃 보드를 연결하여 바이너리 코드를 올리려면 가상의 IP를 통해 USB 장치를 연결해 주는 usbipd-win을 설치해야 한다.

[주] WSL에서 USB 장치를 연결 방법의 자세한 설명은 아래 링크 참조,

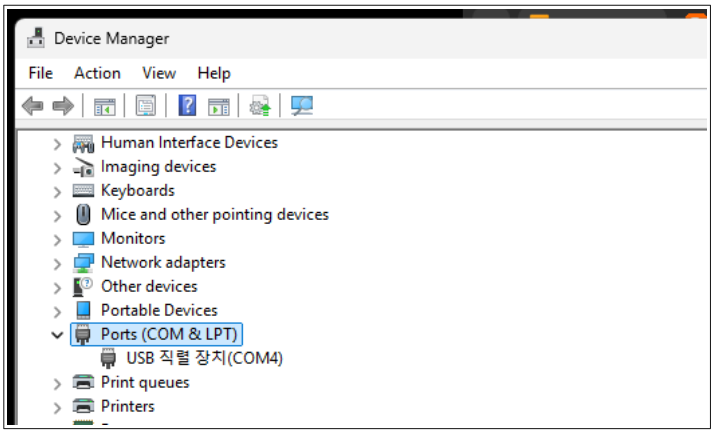
https://learn.microsoft.com/ko-kr/windows/wsl/connect-usb

1) USBIPD-WIN의 최신 버젼 설치 프로그램을 아래 링크에서 내려 받아 실행한다.

https://github.com/dorssel/usbipd-win/releases

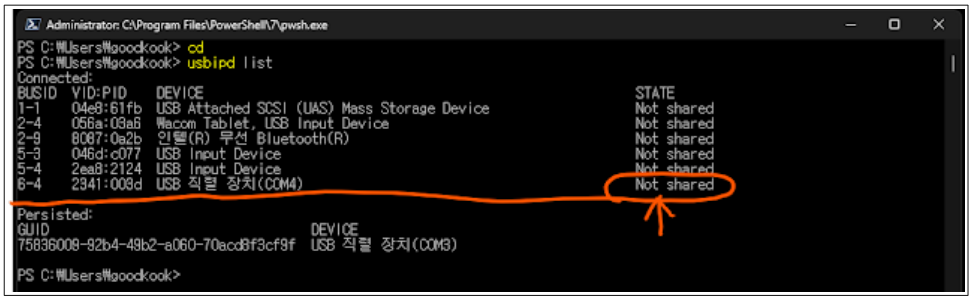


2) Arduino DUE 타깃 보드를 PC의 USB 포트에 연결하면 시리얼 포트로 인식된다.



3) 파워 쉘(윈도우즈 명령 창)을 관리자 권한으로 열어 USBIPD-WIN으로 공유할 수 있는 USB 장치를 검색한다.

```
PS> usbipd list
```



BUSID 6-4 로 할당된 COM4 장치를 윈도우즈와 WSL 사이에 공유 시킨다.

```
PS> usbipd bind --busid 6-4
PS> usbipd list
```

[주] BUSID와 시리얼 포드(COM) 번호는 PC의 상태(USB 허브)에 따라 바뀐다. 공유(bind)는 관리자 권한의 파워쉘 명령창에서 가능하다. 한번 공유된 장치는 다시 공유 시키지 않아도 된다.

4) 공유한 USB 장치(타깃 보드)를 WSL에 연결 한다.

```
PS> usbipd attach --wsl --busid 6-4
```

5) 리눅스 터미널에서 USB 장치들을 확인해 본다.

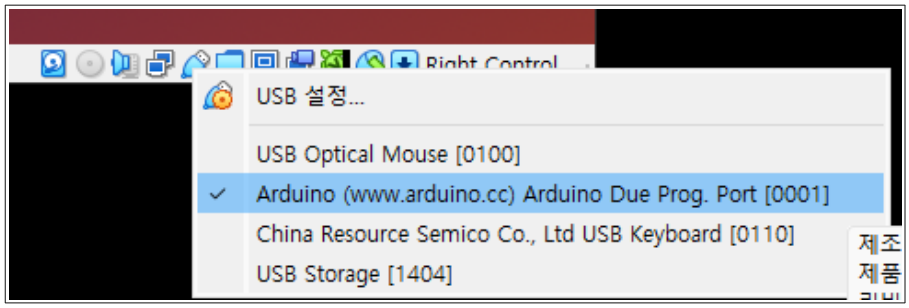
```
$ lsusb
```

아듀이노 DUE 보드의 프로그래밍 포트가 USB 장치로 인식되었다.

b. 타깃 보드(윈도우즈 USB 장치)를 가상 머신 리눅스로 연결

가상 머신에 설치된 리눅스의 경우 USB 장치를 연결을 위해 우회할 필요 없다. 리눅스 머신 그 자체이기 때문이다. 윈도우즈 PC의 USB에 연결된 아듀이노 DUE 보드를 가상 머신(VirtualBox) 리눅스로 인식 시킨다. USB 아이콘에 마우스 오른쪽 버튼을 누르면,

연결 가능한 장치 목록이 나열된다. "Arduino DUE Programing Port"를 선택한다.



c. USB 장치로 접근 권한이 제한 될 때

USB 포트에 연결된 장치(아듀이노 보드와 FPGA 다운로더 케이블 등)는 리눅스에서 대부분 시리얼 포트로 인식된다. 장치 이름이 /dev/ttyACMx 또는 /dev/ttyUSBx 으로 정해지는 경우 일반 사용자에게 접근 권한이 제한되기도 한다.

```
$ ll /dev/ttyACM0
crw-rw---- 1 root dialout 166, 0 Jun 29 17:06 /dev/ttyACM0
```

접근 권한이 있는 그룹이 dialout 다. 만일 사용자가 이 그룹에 속하지 않았을 경우 접근이 제한된다. 해당 장치의 속성을 변경하여 제한된 접근 권한을 풀어 줄 수 있다.

```
$ sudo chmod 666 /dev/ttyACM0
```

장치가 새로 연결될 때마다 이를 풀어주기 번거로울 경우 사용자를 dialout 그룹에 넣어 준다. 현재 그룹의 목록을 보자.

```
$ groups
mychip root adm dialout cdrom sudo dip plugdev users lpadmin
```

현 사용자 mychip 를 dialout 그룹에 속하게 변경 한 후 그룹에 추가한다.

```
$ newgrp dialout
$ sudo adduser mychip dialout
```

사용자 mychip이 속한 그룹을 확인 후 재부팅,

```
$ groups mychip
mychip : mychip root adm dialout cdrom sudo dip plugdev users lpadmin
$ restart
```

2-4. 보드 테스트(Blink 예제)

아듀이노 개발환경이 설치되었다. 구입 한 보드를 간단한 예제 Blink(내장된 LED 깜빡이기)를 가지고 테스트 한다.

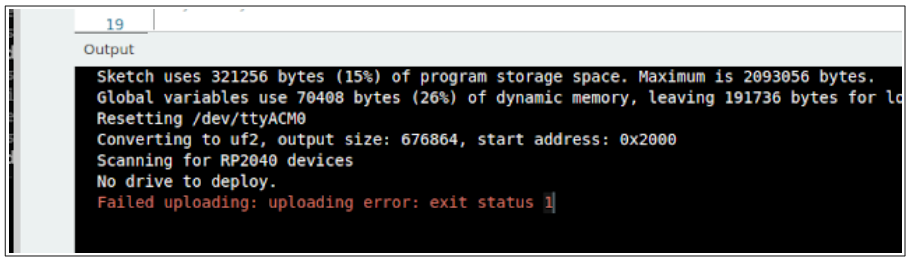
a. Arduino DUE

간단한 응용 프로그램으로 Arduino DUE 보드에 내장된 LED를 깜빡여 본다. Arduino IDE 를 실행 시켜 연결된 타깃 보드(Select Board)를 확인한다.

메뉴 File > Examples > Blink 선택하여 간단한 예제를 불러온 후 컴파일 하고 타깃 보드에 업로드 하면 LED가 깜빡이는 것을 보게 될 것이다.

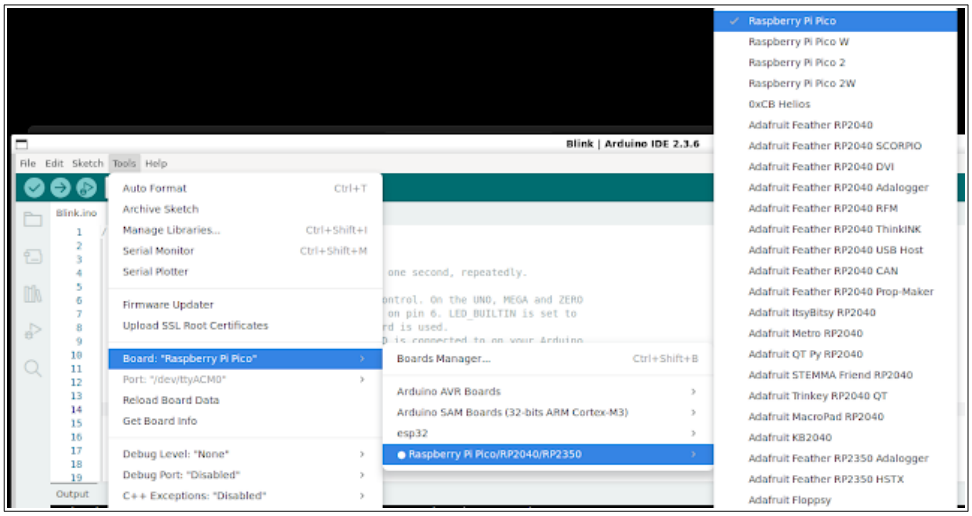
b. Raspberry Pi Pico

라즈베리 파이 "피코"는 아듀이노의 규격을 따르는 보드가 아니다. 컴파일 된 바이너리를 타깃 보드로 올리는 절차가 다르다. 보드 위에 BOOTSEL 버튼을 누른채 부팅 시키면 PC의 이동식 저장 장치로 인식된다. 이 저장 장치에 바이너리 파일(UF2 형식)을 복사해 주면 이를 컨트롤러에 적재하고 재부팅 된다. 아듀이노는 이 과정을 지원하지만 문제는 WSL에 설치된 리눅스다. USB 장치를 직접 연결할 수 없는 WSL은 피코 보드를 BOOTSEL 모드로 전환 시킬 수 있지만 usbipd의 간여 없이 자동 재연결이 불가하다. 윈도우즈 디스크를 재연결 시키지 않아 타깃 보드 찾기에 실패한다.

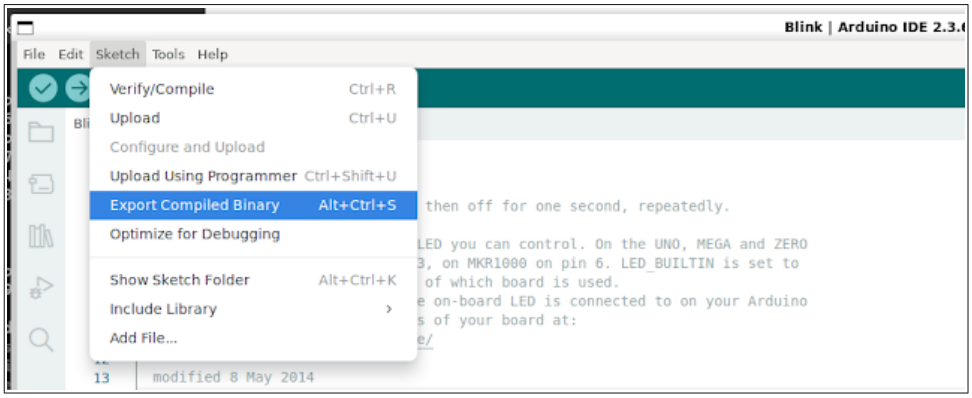


이를 해결할 방법은 아듀이노 환경에서 직접 올리지 않고 디스크 장치에 복사하는 방법이 있다.

1) 타깃 보드를 인식 시킬 필요 없이 강제로 RP2040을 지정한다. 메뉴 Tools > Board: > Raspberry PI: > Raspberry Pi Pico 를 따라가면 타깃 보드를 지정할 수 있다.



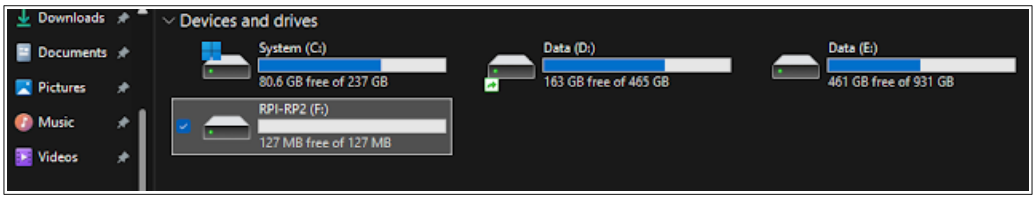
(2) 메뉴 Sketch > Export Compile Binary를 선택하면 타깃 보드에 업로드 하지 않고 파일로 저장한다.



컴파일 한 바이너리 파일(.UF2)은 프로젝트 디렉토리에서 아래로 다음에 위치한다.

```
.../Blink/build/rp2040.rp2040.rpipico/Blink.ino.uf2
```

(3) "피코" 보드를 BOOTSEL 모드로 부팅(BOOTSEL 버튼을 누른채 리셋 또는 전원인가)하면 윈도우즈의 USB 저장장치로 인식된다.



(4) RP2 드라이브를 WSL의 적당한 디렉토리(아래는 _uf2_)에 마운트 한다.

```
$ mkdir ~/_uf2_
$ sudo mount -t drvfs f: ~/_uf2_
```

(5) 위의 (4)에서 생성한 컴파일 바이너리를 ~/_uf2_ 로 복사 한다.

```
$ cp Blink/build/rp2040.rp2040.rpipico/Blink.ino.uf2 ~/_uf2_
```

"피코"보드는 uf2 파일이 복사되는 즉시 재부팅 되어 바이너리를 컨트롤러 메모리에 적재한다. 보드의 LED 가 반짝이는 것을 보게 될 것이다.

c. ESP32-S/C

Espressif Systems 사의 32비트 마이크로 컨트롤러를 채용한 보드다. ESP32 보드는 같은 이름이지만 두개의 다른 코어를 사용한다. C 시리즈는 RISC-V, S 시리즈는 Tensilica Xtensa 코어를 사용한 SoC다. 제조사에서 새 프로젝트에 C 시리즈를 추천하고 있다. 두 코어는 기계어 수준의 호환성이 없다. 서로 다른 코어지만 높은 수준의 C/C++언어 개발 환경에서 이를 구분하지 않는다. 명령 체계가 다른 코어지만 입출력 장치들의 호환성을 가지고 있기 때문이다. 높은 추상화 수준의 언어로 작성된 코드들은 각각의 언어 변환기(컴파일러)로 다른 CPU에 이식 될 수 있다. 이식성(portablity)은 C/C++ 언어의 특장점이다. 이식성의 걸림돌은 입출력 장치의 호환성이다. 하드웨어 구조상

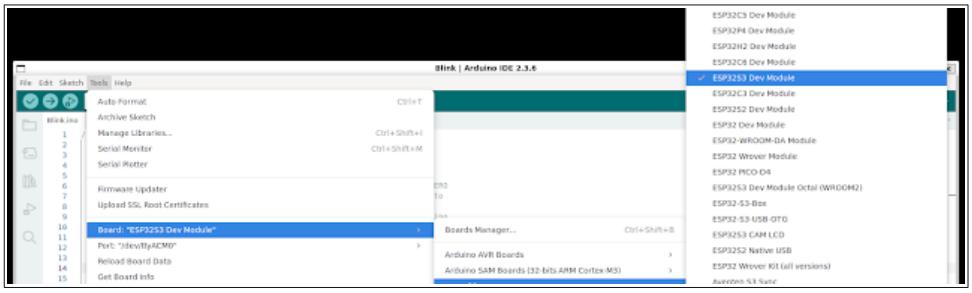
호환성이 없다면 중간에 위치한 장치 구동 라이브러리로 이를 보완 할 수 있다. 앞의 예제 Blink를 ESP32-S3 코어를 채택한 보드에서 실행 시켜본다.

1) ESP32-S3 보드를 PC의 USB에 꼽으면 시리얼 포트로 인식된다.

2) WSL 사용자라면 usbipd 를 통해 리눅스로 공유 후 연결되도록 한다.

```
PS > usbipd list
Connected:
BUSID  VID:PID    DEVICE                                        STATE
6-2    1a86:55d3  USB-Enhanced-SERIAL CH343(COM6)               Not shared
PS> usbipd bind --busid 6-2
PS> usbipd attach --wsl --busid 6-2
```

3) ESP32 타깃 보드는 아듀이노 환경에 잘 어울린다. 메뉴 Tools > Board > ESP32: 를 따라간 후 타깃 보드를 선택한다. C 또는 S 시리즈를 비롯하여 다양한 보드들이 있다.



4) 타깃 보드가 선정되면 예제의 소스 코드에 수정 없이 컴파일 되어 바이너리를 올릴 수 있다.

앞서 시험해 본 타깃 보드(Arduino DUE, Raspberry Pi Pico)의 프로세서 코어는 모두 32비트 ARM이었다. 동일한 코드를 ESP32 프로세서 코어를 채택한 보드에서 실행 될 수 있다. 프로세서 코어는 물론 세가지 보드들의 구성(회로도)도 다르지만 사용자들에 의해 호환성을 유지 할 수 있는 라이브러리 들을 개발하여 나눔으로써 아듀이노는 하드웨어와 소프트웨어를 모두 아우른 개발 플랫 폼이 되었다. 물론 매우 단순한 예제이긴 하지만 이런 이식성(portablity)은 아듀이노 개발 플랫폼과 라이브러리 제작에 기여한 집단 지성의 성과다.

2-5. Arduino CLI

통합 환경은 도구 사용을 직관적이게 해주지만 번거롭다. 아듀이노는 명령 줄에서 실행 할 수 있는 CLI(Command-Line Interface)를 제공한다. 스크립트를 사용하면 IDE에서 여러 차례 버튼을 누르고 명령마다 옵션을 선택하는 번거로움을 피할 수 있다. CLI는 그 외 다양한 장점이 있다.

[주] Arduino CLI and the art of command line,
https://www.youtube.com/watch?app=desktop&v=cVod8k713_8

아듀이노 명령줄 사용법은 아래 링크를 참고한다.

[주] Arduino-CLI Getting Started,
https://arduino.github.io/arduino-cli/1.3/getting-started/

앞서 통합 환경에서 마이크로 컨트롤러(코어)와 타깃 보드 용 개발 도구들을 모두 설치한 상태다. 아듀이노 CLI는 IDE에서 설치했던 각종 라이브러리와 보드 도구들을 공유하므로 재설치 할 필요 없다. 예제 Blink 를 CLI 환경에서 수행해본다.

a. Arduino DUE

작업 디렉토리로 이동,

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-13_Arduino_Blink
```

예제의 Makefile 을 작성해 두었다.

```
$ make
Arduino-CLI
Usage:
   make help
   make config
   NEW_SKETCH=[MyNewSketsh] make new_sketch
   FQBN=[board name] SKETCH=[sketch name] make compile
   PORT=[/dev/ttyACM0] FQBN=[board name] SKETCH=[sketch name] make upload
   make board_list
   make core_list
   BOARD_NAME=[name] make board_listall
   CORE_NAME=[name] make core_list
   make update_cache
   make core_update
   CORE_NAME=[name] make install_core
   LIB_NAME=[libname] make lib_search
   LIB_NAME=[libname] make lib_install
```

먼저 어떤 보드가 연결되었는지 확인한다.

```
$ make board_list
arduino-cli board list
Port         ..... FQBN                              Core
/dev/ttyACM0 ..... arduino:sam:arduino_due_x_dbg  arduino:sam
```

현재 연결된 보드의 풀-네임(FQBN, Fully Quarified Board Name)과 코어를 알 수 있다. 이를 인수로 주고 예제 Blink 를 컴파일 한다.

```
$ FQBN=arduino:sam:arduino_due_x_dbg SKETCH=Blink make compile
arduino-cli compile --fqbn arduino:sam:arduino_due_x_dbg Blink
Sketch uses 10692 bytes (2%) of program storage space. Maximum is 524288
bytes.
```

컴파일이 완료 되었으면 타깃 보드에 바이너리를 올려보자.

```
$ PORT=/dev/ttyACM0 FQBN=arduino:sam:arduino_due_x_dbg SKETCH=Blink make upload
arduino-cli upload -p /dev/ttyACM0 --fqbn arduino:sam:arduino_due_x_dbg Blink
Atmel SMART device 0x285e0a60 found
```

```
Erase flash
done in 0.039 seconds
Write 11872 bytes to flash (47 pages)
[==============================] 100% (47/47 pages)
done in 2.818 seconds
Set boot flash true
CPU reset.
New upload port: /dev/ttyACM0 (serial)
```

b. Raspberry Pi Pico

"피코" 보드에 바이너리를 올리려면 통합 환경에서 나와 몇가지 절차가 필요했다(WSL 인 경우에 한함). 이 절차를 Makefile에 기술하였다. 타깃 보드가 연결되었는지 보자.

```
$ make board_list
```

```
arduino-cli board_list
Port          ... ... FQBN                       Core
/dev/ttyACM0  ... ... rp2040:rp2040:degz_suibo rp2040:rp2040
                      rp2040:rp2040:evn_alpha  rp2040:rp2040
                      rp2040:rp2040:generic    rp2040:rp2040
                      rp2040:rp2040:sea_picro  rp2040:rp2040
                      rp2040:rp2040:rpipico    rp2040:rp2040
```

포트에 한개의 보드가 꼽혀 있지만 여러가지 보드들이 나열된다. 모두 호환되는 목록들이다. FQBN을 변수로 주고 컴파일 한다.

```
$ FQBN=rp2040:rp2040:generic SKETCH=Blink make compile
```

```
arduino-cli compile --fqbn rp2040:rp2040:generic Blink
Sketch uses 58832 bytes (2%) of program storage space.
 Maximum is 2093056 bytes.
Global variables use 9640 bytes (3%) of dynamic memory,
 leaving 252504 bytes for local variables.
Maximum is 262144 bytes.
```

"피코" 보드에 컴파일 된 바이너리를 업로드 하자.

```
$ PORT=/dev/ttyACM0 FQBN=rp2040:rp2040:generic SKETCH=Blink make upload
```

```
arduino-cli upload -p /dev/ttyACM0 --fqbn rp2040:rp2040:generic Blink
Resetting /dev/ttyACM0
Converting to uf2, output size: 151040, start address: 0x2000
Scanning for RP2040 devices
No drive to deploy.
Failed uploading: uploading error: exit status 1
```

업로드를 실패했다. 그 이유는 앞서 설명한 대로다. 컴파일한 후 바이너리를 따로 저장하는 옵션은 -e 다. Makefile에 "피코"를 위한 타깃을 아래와 같이 작성하였다.

```
compile_pico:
    arduino-cli compile --fqbn $(FQBN) $(SKETCH) -e
```

바이너리를 "피코" 보드에 올리기 전에 RP2 디스크를 먼저 마운트 후 바이너리를 복사한다.

```
upload_pico: $(SKETCH)/build/rp2040.rp2040.generic/$(SKETCH).ino.uf2
    sudo mount -t drvfs f: ~/_uf2_
    cp $(SKETCH)/build/rp2040.rp2040.rpipicow/$(SKETCH).ino.uf2 ~/_uf2_
```

c. ESP32

ESP32 보드는 아듀이노 환경과 잘 어울린다. 보드 연결을 확인해 보자.

```
$ make board_list

arduino-cli board list
Port         Protocol Type              Board Name FQBN Core
/dev/ttyACM0 serial   Serial Port (USB) Unknown
```

아쉽게도 보드 이름을 확인 하지 못한다. 하지만 보드 이름을 알고 있으므로 FQBN 만 얻으면 된다. 설치된 도구의 지원 목록에서 esp32s3 를 찾아본다.

```
$ BOARD_NAME=esp32s3 make board_listall

arduino-cli board listall esp32s3
Board Name              FQBN
…
ESP32-S3 PowerFeather   esp32:esp32:esp32s3_powerfeather
ESP32-S3-USB-OTG        esp32:esp32:esp32s3usbotg
ESP32S3 Dev Module      esp32:esp32:esp32s3
```

ESP32 S3 보드의 FQBN을 알아냈다. 컴파일 해보자.

```
$ FQBN=esp32:esp32:esp32s3 SKETCH=Blink make compile

arduino-cli compile --fqbn esp32:esp32:esp32s3 Blink
Sketch uses 315247 bytes (24%) of program storage space.
 Maximum is 1310720 bytes.
Global variables use 20568 bytes (6%) of dynamic memory,
 leaving 307112 bytes for local variables. Maximum is 327680 bytes.
```

보드에 바이너리 올리기,

```
$ PORT=/dev/ttyACM0 FQBN=esp32:esp32:esp32s3 SKETCH=Blink make upload

arduino-cli upload -p /dev/ttyACM0 --fqbn esp32:esp32:esp32s3 Blink

esptool v5.0.0
Connected to ESP32-S3 on /dev/ttyACM0:
Chip type: ESP32-S3 (QFN56) (revision v0.1)
Features: Wi-Fi, BT 5 (LE), Dual Core + LP Core, 240MHz
Crystal frequency:  40MHz
MAC:                f4:12:fa:ce:e4:60
…
New upload port: /dev/ttyACM0 (serial)
```

2-6. "남의 칩"에서 FIR 필터 알고리즘 실행시키기

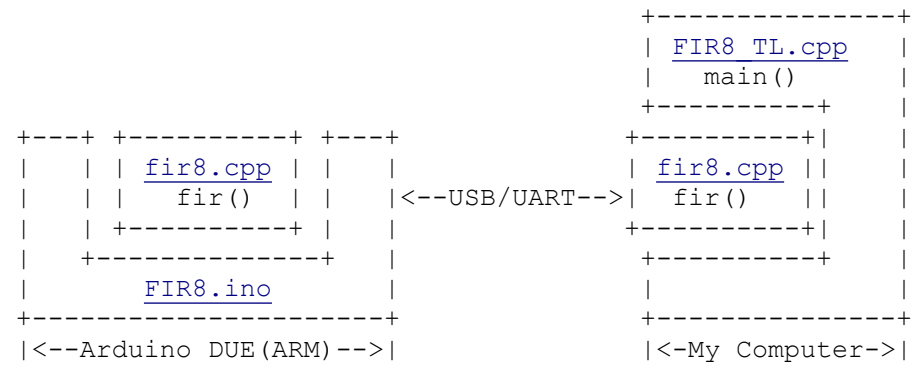
LED 깜빡이기는 너무 단순했다. FIR 필터 알고리즘[연구노트9]을 "내 칩"으로 설계하기 전에 "남의 칩"에서 실행시켜보자.

a. 개요

FIR 필터 알고리즘을 검증하기 위해 C++로 테스트 벤치를 작성했었다[연구노트9]. FIR 알고리즘을 "내 컴퓨터"에서 실행하기 위한 테스트 벤치는 다음과 같다. FIR 필터 함수 fir()를 기술한 fir8.cpp 를 테스트 벤치 fir8_tb.cpp 와 함께 컴파일 하고 실행 하면 "내 컴퓨터"에서 알고리즘을 시험하게 된다.

```
// Filename: .../c_untimed/fir8_tb.cpp
int main(void)
{
    data_t  x[F_SAMPLE];
    acc_t   y[F_SAMPLE], yn;
    ...…
    for (int t=0; t<F_SAMPLE; t++)
    {
        ...…
        fir(&yn, x[t]); // FIR Filter
        ...…
    }
    ...…
    return 0;
}
```

이번에는 fir() 를 "남의 칩", 아듀이노 DUE 보드에 장착된 ARM 코어에서 실행 되도록 옮겨보자.



알고리즘 fir()은 "남의 칩(Arduino DUE보드의 ARM)"에서 실행될 것이며 "내 컴퓨터"는 USB의 시리얼 통신 포트를 통해 입력을 주고 출력을 받아온다. "남의 칩"에서 FIR 필터 알고리즘 fir()의 수행을 위해 시리얼 통신 인터페이스를 감싼 아듀이노 코드 FIR8.ino는 다음과 같다. 시리얼 통신으로 받은 값을 입력으로 FIR 필터 함수 fir()를 호출하고 계산된 값을 내보낸다.

```
// Filename: .../emulation/FIR8.ino
#include <Arduino.h>
#include "fir8.h"
void setup()
{
  // start serial port
  Serial.begin(9600);
  while (!Serial);
  establishContact();
}
```

```
void loop()
{
    data_t  x;  // unsigned 8-bit
    acc_t   y;  // unsigned 16-bit
    ...…
    // Receive Emulator/DUT in-vector from HOST
    if (Serial.available() >= N_RX)
    {
        x = Serial.read();
        if (Serial.availableForWrite() >= N_RX)
            Serial.write((data_t)x); // Loop-Back
    }
    else    return;

    fir( &y, x);  // FIR Filter @ fir8.cpp

    if (Serial.availableForWrite() >= N_TX)
    {
        Serial.write((data_t)(y>>0) & 0x0FF); // LSB
        Serial.write((data_t)(y>>8) & 0x0FF); // MSB
    }
    ...…
}
```

"내 컴퓨터"의 테스트 벤치 FIR8_TL.cpp 는 위의 fir8_tb.cpp와 동일하다. 외견상 함수 fir() 이 "남의 칩"에서 실행 되는지 모른다. 알 필요도 없다. "내 컴퓨터"의 fir() 함수는 아래와 같다. 시리얼 통신 포트를 통해 "남의 칩"으로 보내고 결과를 받아올 뿐이다.

```
// Filename: .../emulation/fir8.cpp
bool fir(acc_t* Yout, data_t Xin)
{
    uint8_t     _tx, _rx;
    uint16_t    _Yout;
    //-----------------------------------------------------------
    // Connect Arduino via Serial Port
    //-----------------------------------------------------------
    static int fd = 0; // Serial port file descriptor
    if (fd<=0)
    {
         ......
    }
    //-------------------------------------------------------------------
    // Send to Emulator
    //-------------------------------------------------------------------
    _tx = Xin;
    while(write(fd, &_tx, 1)<=0)  usleep(10);
    while(read(fd, &_rx, 1)<=0)   usleep(10);  // Loop-Back
    if (_tx != _rx) // Loop-Back Test
    {
        fprintf(stderr, "COMM Error......\n");
        return false;
    }
    //-------------------------------------------------------------------
    // Receive from Emulator
    //-------------------------------------------------------------------
```

```
    while(read(fd, &_rx, 1)<=0)   usleep(10);
    _Yout = (uint16_t)_rx;
    while(read(fd, &_rx, 1)<=0)   usleep(10);
    _Yout |= ((uint16_t)_rx<<8);
    *Yout = _Yout;
    return true;
}
```

b. 실습

"남의 칩"에서 FIR 필터 알고리즘 실행시키기의 예제 디렉토리로 이동,

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-15_Lab6_FIR8_c_untimed_Arduino
$ tree
.
├── emulation
│   ├── fir8.cpp
│   ├── fir8.h
│   ├── FIR8_TL.cpp
│   └── sc_timed
│       ├── Makefile
│       ├── sc_fir8.gtkw
│       ├── sc_fir8.h
│       ├── sc_fir8.sav
│       ├── sc_fir8_tb.cpp
│       ├── sc_fir8_tb.h
│       └── sc_main.cpp
├── FIR8
│   ├── fir8.cpp -> ../../2-5_Lab3_FIR8/c_untimed/fir8.cpp
│   ├── fir8.h -> ../../2-5_Lab3_FIR8/c_untimed/fir8.h
│   └── FIR8.ino
├── Makefile
└── sc_plotDFT.py
```

아듀이노 DUE 용 바이너리 빌드,

```
$ make build
arduino-cli compile --clean \
    --fqbn arduino:sam:arduino_due_x_dbg FIR8 \
    --build-property compiler.cpp.extra_flags=-DFIR_SHIFTER_VERSION
Sketch uses 10964 bytes (2%) of program storage space. Maximum is
524288 bytes.
```

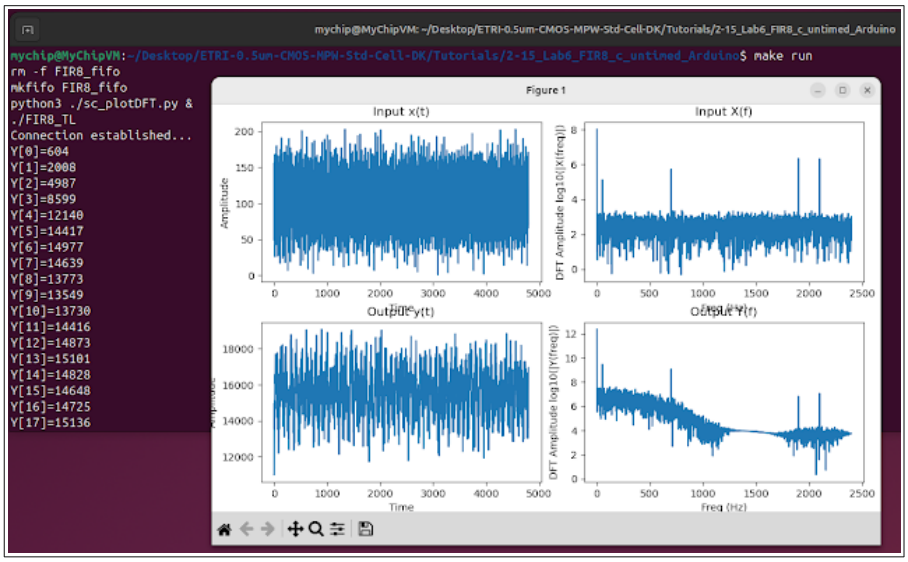
컴파일된 바이너리 업로드,

```
$ make upload
arduino-cli upload \
    -p /dev/ttyACM0 \
    --fqbn arduino:sam:arduino_due_x_dbg FIR8
Atmel SMART device 0x285e0a60 found
Erase flash
done in 0.042 seconds
Write 12144 bytes to flash (48 pages)
[==============================] 100% (48/48 pages)
```

```
done in 2.860 seconds
Set boot flash true
CPU reset.
New upload port: /dev/ttyACM0 (serial)
```

"남의 칩"에서 실행,

```
$ make run
```



3. Vitis-HLS

Vitis와 Vivado 는 자일링스(Xilinx) FPGA 개발 도구를 모아놓은 패키지다. 자사의 FPGA에 특화된 합성과 배치-배선 도구 뿐만 아니라 다양한 설계 자동화 도구들로 구성되었다. 특히 Vitis-HLS 는 고위 합성(HLS, High-Level Systhesis) 도구다. 표준 C/C++ 구문을 베릴로그 RTL 로 변환(합성)해 준다. 변환된 베릴로그 RTL 코드는 범용성을 가지고 있어서 다른 FPGA 뿐만 아니라 ASIC 용으로도 사용할 수 있다. 여기에는 "내 칩 제작 서비스"의 표준 셀 기반 디자인 킷을 적용 할 수 있다. 자일링스의 도구들은 몇가지 기능(합성 후 분석도구 등)을 제외하고 무료 사용 할 수 있다.

3-1. Vitis-HLS 설치

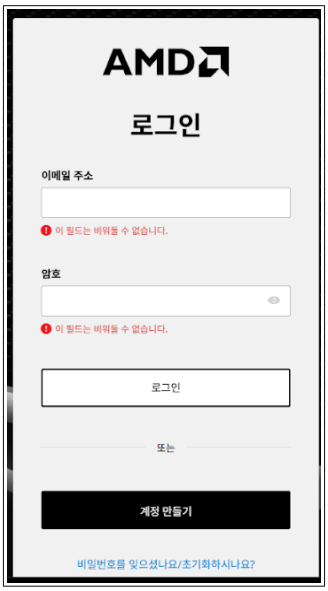
자일링스 도구 내려받기 사이트의 링크는 아래와 같다.

https://www.xilinx.com/support/download.html

리눅스 설치 파일을 내려받는다.

AMD Unified Installer for FPGAs & Adaptive SoCs 2025.1: Linux Self Extracting Web Installer (BIN - 335.53 MB)
https://www.xilinx.com/member/forms/download/xef.html?filename=FPGAs_AdaptiveSoCs_Unified_SDI_2025.1_0530_0145_Lin64.bin

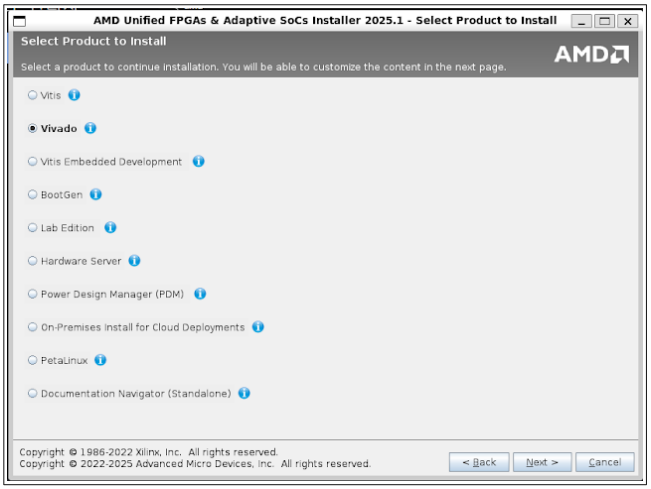
설치 파일을 내려받기 위해 AMD의 계정이 필요하다. AMD 계정이 없다면 새로 만들도록 한다. 신청서를 성실히 채워 넣으면 즉시 확인 메일이 올 것이다.



내려받은 설치 파일을 실행 한다. 내려받은 파일에 실행 속성이 없을 경우 다음과 같이 파일 속성을 변경한 후 관리자로 실행한다.

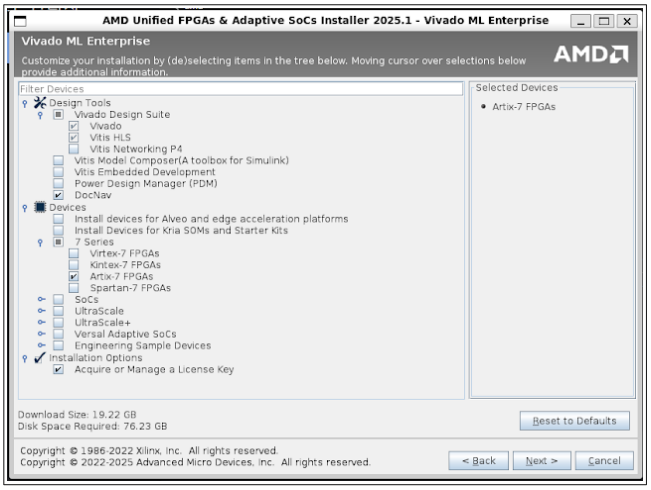
```
$ chmod +x FPGAs_AdaptiveSoCs_Unified_SDI_2025.1_0530_0145_Lin64.bin
$ sudo ./FPGAs_AdaptiveSoCs_Unified_SDI_2025.1_0530_0145_Lin64.bin
```

자일링스 도구들은 여러가지 형식으로 패키지되어 있다. 최신의 FPGA 제품군을 사용하려면 Vitis를 선택한다. Vitis 를 선택하면 설치 용량이 매우 커진다. Vivado 패키지에 VitisHLS도 포함되어 있다.

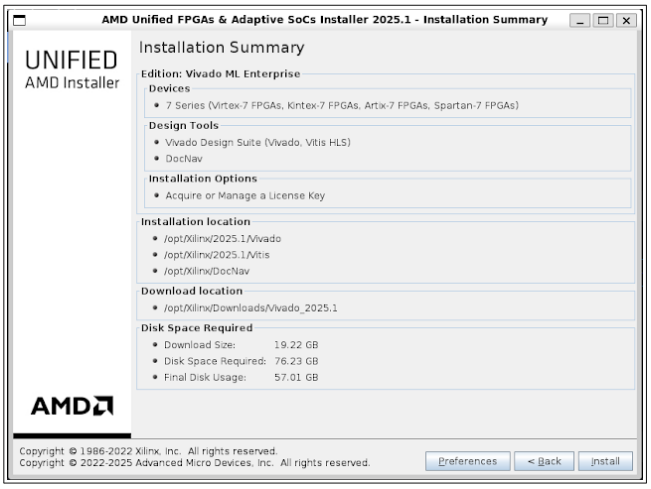


표준 보다는 전문가 답게 기업형으로 설치한다. 표준형 도구에는 VitisHLS가 빠져있다. 기업형이라 해서 사용료를 요구하고 그러지 않는다.

설계 도구로 Vivado와 함께 VitisHLS 선택, FPGA 제품군으로 Artix-7만 선택해도 좋다. 여러 제품군을 고를 수록 설치 용량이 거대해 진다.



선택했던 내용을 살펴보고 설치를 시작하자. 설치할 디렉토리 위치를 /opt/Xilinx 로 했다.



무려 20GByte 정도를 내려받아 설치 용량은 57GByte 다. 인터넷 속도에 달렸지만 족히 한 시간은 걸릴 것이니 참고 기다리다 보면 설치가 완료된다. 자일링스 도구들을 사용하려면 로케일을 맞춰준다.

```
$ sudo locale-gen en_US.UTF-8
```

3-2. FIR 필터 알고리즘의 HLS

HLS(High-Level Synthesis)은 다음편에 배우기로 하고 HLS 맛보기다. C/C++로 작성한 FIR 필터[연구노트9]를 RTL 로 합성해보자. 고위합성의 최근 경향과 안내는 아래 링크를 참조한다.

[주] 고위합성 자동화 도구의 동향(Automatic High-Level Code Deployment)
https://hls-goodkook.blogspot.com/2023/10/automatic-high-level-code-deployment.html
[주] 고위합성 튜토리얼
https://hls-goodkook.blogspot.com/2021/08/ug871-xilinx-high-level-synthesis.html

고위합성 예제 디렉토리,

```
$ cd ~/ETRI050_DesignKit/Tutorials/2-8_Lab6_FIR8_c_untimed_Vitis-HLS
$ tree
.
├── /emulation          :  FPGA 에뮬레이션
├── /ETRI050            :  "내 칩 서비스" ASIC
├── /simulation         :  RTL/SystemC 시뮬레이션
│   ├── Makefile
│   ├── fir_TL_Bulk.cpp
│   ├── fir_TL.cpp
│   ├── sc_fir_TB.cpp
│   ├── sc_fir_TB.h
│   └── sc_main.cpp
├── Makefile
├── fir.cfg              : HLS 합성 조건
└── Vitis-HLS.tcl        : VitisHLS 스크립트
```

예제 디렉토리에 Makefile 이 준비 되었다.

```
$ make
Vitis-HLS Project: fir
   make csynth
   make view_rpt
   make co-sim
   make co-emu
   make clean
   * command-line variable HW_STYLE must be set before build
         HW_STYLE=[MACC | SHIFT | ARRAY] make csynth
```

a. 고위합성

C 로 작성한 언타임드 FIR 필터 fir8.cpp [연구노트9]를 가져다 고위 합성 한다.

```
$ make csynth
```

```
 vitis-run --mode hls --tcl Vitis-HLS.tcl
 ****** vitis-run v2024.2 (64-bit)
   **** SW Build 5239630 on 2024-11-10-11:19:46
   **** Start of session at: Mon Aug 18 16:37:10 2025
     ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
     ** Copyright 2022-2024 Advanced Micro Devices, Inc. All Rights Reserved.
   **** HLS Build v2024.2 5238294
```

로그를 보면서 고위 합성이 진행되는 과정을 간략히 살펴보자.

```
Sourcing Tcl script '...Tutorials/2-8_Lab6_FIR8_c_untimed_Vitis-HLS/Vitis-HLS.tcl'
-DFIR_MAC_VERSION
```

언타임드 C 소스 코드의 합성 가능성을 분석하고 RTL로 변환 시작,

```
INFO: [HLS 200-1510] Running: csynth_design
INFO: [HLS 200-10] Analyzing design file
    '../2-5_Lab3_FIR8/c_untimed/fir8.cpp' ...
INFO: [HLS 214-376] automatically set the pipeline for Loop< SHIFTER_LOOP>
    at ../2-5_Lab3_FIR8/c_untimed/fir8.cpp:56:5
INFO: [HLS 214-376] automatically set the pipeline for Loop< MACC_LOOP>
    at ../2-5_Lab3_FIR8/c_untimed/fir8.cpp:66:5
INFO: [HLS 200-10] Starting code transformations …
INFO: [HLS 200-10] Checking synthesizability …
INFO: [HLS 200-10] Starting hardware synthesis …
INFO: [HLS 200-10] Synthesizing 'fir' ...
```

반복문 내에 파이프라인 스케쥴링과 자원 공유(resource sharing) 가능성 분석,

```
INFO: [HLS 200-10]
----------------------------------------------------------------
INFO: [HLS 200-42] -- Implementing module 'fir_Pipeline_SHIFTER_LOOP'
INFO: [HLS 200-10]
----------------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling …
INFO: [SCHED 204-61] Pipelining loop 'SHIFTER_LOOP'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA,
                     Final II = 1, Depth = 2, loop 'SHIFTER_LOOP'
INFO: [BIND 205-100] Starting micro-architecture generation …
INFO: [BIND 205-101] Performing variable lifetime analysis.
INFO: [BIND 205-101] Exploring resource sharing.
INFO: [BIND 205-101] Binding ...
```

반복문 구조에서 클럭 소요(iteration)의 목표 클럭 수에 맞춰(예에서 목표 클럭 수 지정은 없음), 초기 지연(II) 1 클럭, 반복 준비(Depth)에 5클럭이 소요되도록 합성되었다(시뮬레이션 결과 파형 참조).

```
INFO: [HLS 200-10]----------------------------------------------------------
INFO: [HLS 200-42] -- Implementing module 'fir_Pipeline_MACC_LOOP'
INFO: [HLS 200-10]----------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'MACC_LOOP'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA,
                     Final II = 1, Depth = 5, loop 'MACC_LOOP'
INFO: [BIND 205-100] Starting micro-architecture generation ...
```

```
INFO: [BIND 205-101] Performing variable lifetime analysis.
INFO: [BIND 205-101] Exploring resource sharing.
```

RTL 하드웨어 코드 생성(합성 가능 베릴로그 및 VHDL),

```
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [HLS 200-42] -- Implementing module 'fir'
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [HLS 200-10] -- Generating RTL for module 'fir_Pipeline_SHIFTER_LOOP'
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [HLS 200-10] -- Generating RTL for module 'fir_Pipeline_MACC_LOOP'
INFO: [HLS 200-10]-------------------------------------------------------
```

합성된 RTL 하드웨어는 fir() 수행에 여러 클럭이 소요되므로 입출력 신호 핸드쉐이크(인터페이스 합성),

```
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [HLS 200-10] -- Generating RTL for module 'fir'
INFO: [HLS 200-10]-------------------------------------------------------
INFO: [RTGEN 206-500] Setting interface mode on port 'fir/y' to 'ap_vld'.
INFO: [RTGEN 206-500] Setting interface mode on port 'fir/x' to 'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'fir' to 'ap_ctrl_hs'.
```

고위합성 후 생성된 RTL은 ./fir/hls_component/syn 폴더에 저장되었다.

```
~/ETRI050_DesignKit/Tutorials/2-8_Lab6_FIR8_c_untimed_Vitis-HLS
├── /emulation
├── /ETRI050
├── /simulation
├── /logs
│   └── hls_run_tcl.log
├── /fir
│   ├── hls.app
│   └── /hls_component
│       ├── hls_component.log
│       ├── /impl
│       │   ├── /verilog
│       │   └── /vhdl
│       └── /syn
│           ├── /report
│           │   ├── csynth_design_size.rpt
│           │   ├── csynth.rpt
│           │   ├── fir_csynth.rpt
│           │   ├── fir_Pipeline_MACC_LOOP_csynth.rpt
│           │   └── fir_Pipeline_SHIFTER_LOOP_csynth.rpt
│           ├── /verilog
│           │   ├── fir_fir_Pipeline_MACC_LOOP_filter_taps_ROM_AUTO_1R.dat
│           │   ├── fir_fir_Pipeline_MACC_LOOP_filter_taps_ROM_AUTO_1R.v
│           │   ├── fir_fir_Pipeline_MACC_LOOP.v
│           │   ├── fir_fir_Pipeline_SHIFTER_LOOP.v
│           │   ├── fir_flow_control_loop_pipe_sequential_init.v
│           │   ├── fir_mac_muladd_8ns_6ns_16ns_16_4_1.v
│           │   ├── fir_shift_reg_RAM_AUTO_1R1W.dat
```

```
│           │   ├── fir_shift_reg_RAM_AUTO_1R1W.v
│           │   └── fir.v
│           └── vhdl
├── fir.cfg
├── Makefile
└── Vitis-HLS.tcl
```

b. HLS 결과 보기

```
$ make view_rpt
================================================================
== Vitis HLS Report for 'fir'
================================================================
* Project:        fir
* Solution:       hls_component (Vivado IP Flow Target)
* Product family: aartix7
* Target device:  xa7a100t-csg324-2I
```

자일링스의 FPGA를 기준으로 작성된 보고서 이지만 소요 클럭과 연산회로, 플립플롭의 갯수등을 살펴볼 수 있다.

```
================================================================
== Performance Estimates
================================================================
+ Timing:
* Summary:
+--------+---------+----------+------------+
|  Clock |  Target | Estimated| Uncertainty|
+--------+---------+----------+------------+
|ap_clk  |  1.00 us|  4.570 ns|     0.27 us|
+--------+---------+----------+------------+
```

구현 목표를 타 FPGA나 ASIC으로 전환 하는 경우 소요 클럭 수는 유의미 하다.

```
+ Latency:
* Summary:
+---------+---------+-----------+-----------+-----+-----+---------+
|  Latency (cycles) |   Latency (absolute)  |  Interval | Pipeline|
|   min   |   max   |    min    |    max    | min | max |   Type  |
+---------+---------+-----------+-----------+-----+-----+---------+
|       27|       27|  27.000 us|  27.000 us|   28|   28|       no|
+---------+---------+-----------+-----------+-----+-----+---------+
```

반복문에 대하여 상세 레이턴시 클럭수는 다음과 같다. fir8.cpp 에서 매크로 FIR_MAC_VERSION가 적용된 함수 fir()의 계산에 두번의 for-반복문이 있다. SHIFTER_LOOP에 10클럭, MACC_LOOP은 13 클럭을 쓰고 있다.

```
+ Detail:
* Instance:
+--------------------------+--------+------+-----+-----+--------------------+
|                          |Latency(cycles)|  Interval |  Pipeline          |
|          Module          |  min   | max  | min | max |    Type            |
+--------------------------+--------+------+-----+-----+--------------------+
|fir_Pipeline_SHIFTER_LOOP |      10|    10|    9|    9|loop auto-rewind stp|
```

```
|fir_Pipeline_MACC_LOOP     |       13|    13|    9|    9|loop auto-rewind stp|
+---------------------------+---------+------+-----+-----+--------------------+
```

하드웨어 사용량(자일링스 FPGA 기준),

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |        0|    1|     112|    205|    -|
|Memory           |        0|    -|      16|      1|    0|
|Multiplexer      |        -|    -|       0|     78|    -|
|Register         |        -|    -|      15|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |        0|    1|     143|    284|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      270|  240|  126800|  63400|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |        0|   ~0|      ~0|     ~0|    0|
+-----------------+---------+-----+--------+-------+-----+
```

MACC_LOOP 심볼의 반복문 내의 곱셈기는 Xilinx FPGA DSP 블록으로 구현되었다.

```
+ Detail:
* Instance:
+--------------------------+---------+----+-----+-----+-----+
|          Module          | BRAM_18K| DSP|  FF | LUT | URAM|
+--------------------------+---------+----+-----+-----+-----+
|fir_Pipeline_MACC_LOOP    |        0|   1|  100|  115|    0|
|fir_Pipeline_SHIFTER_LOOP |        0|   0|   12|   90|    0|
+--------------------------+---------+----+-----+-----+-----+
| Total                    |        0|   1|  112|  205|    0|
+--------------------------+---------+----+-----+-----+-----+
```

인터페이스 합성 결과는 아래와 같다. C 의 함수 호출을 RTL로 합성하면 클럭 동기에 맞춰 외부와 핸드 쉐이크 신호가 필요하다.

```
================================================================
== Interface
================================================================
* Summary:
+----------+-----+-----+------------+--------------+--------------+
| RTL Ports| Dir | Bits|  Protocol  | Source Object|    C Type    |
+----------+-----+-----+------------+--------------+--------------+
|ap_clk    |   in|    1| ap_ctrl_hs|           fir| return value|
|ap_rst    |   in|    1| ap_ctrl_hs|           fir| return value|
|ap_start  |   in|    1| ap_ctrl_hs|           fir| return value|
|ap_done   |  out|    1| ap_ctrl_hs|           fir| return value|
|ap_idle   |  out|    1| ap_ctrl_hs|           fir| return value|
|ap_ready  |  out|    1| ap_ctrl_hs|           fir| return value|
```

```
|y          |  out|   16|      ap_vld|             y|       pointer|
|y_ap_vld   |  out|    1|      ap_vld|             y|       pointer|
|x          |   in|    8|     ap_none|             x|        scalar|
+----------+-----+-----+------------+--------------+--------------+
```

c. RTL-SystemC 시뮬레이션

HLS로 생성된 베릴로그를 SystemC 테스트벤치에 물려 시뮬레이션 한다.

```
$ make co-sim
make -C simulation run
make[1]: Entering directory '……/2-8_Lab6_FIR8_c_untimed_Vitis-HLS/simulation'
verilator --sc --pins-sc-uint \
               -Wno-WIDTHTRUNC -Wno-WIDTHEXPAND \
               --trace --timing --top-module fir  --exe --build \
               -CFLAGS -g -CFLAGS -I../../../2-5_Lab3_FIR8/c_untimed \
               -CFLAGS -I../../emulation -CFLAGS -DVCD_TRACE_TEST_TB \
               -CFLAGS -DVCD_TRACE_DUT_VERILOG \
               -CFLAGS -DSC_INCLUDE_FX \
               -LDFLAGS -lm -LDFLAGS -lgsl -CFLAGS -DFIR_MAC_VERSION \
               ../fir/hls_component/syn/verilog/*.v \
               ./sc_fir_TB.cpp ./sc_main.cpp \
               ../../2-5_Lab3_FIR8/c_untimed/fir8.cpp \
               ../../2-5_Lab3_FIR8/c_untimed/cnoise.cpp

    ......

make[2]: Leaving directory '……/2-8_Lab6_FIR8_c_untimed_Vitis-HLS/simulation/obj_dir'
- V e r i l a t i o n   R e p o r t: Verilator 5.039 ......
- Verilator: Built from 0.325 MB sources in 9 modules, ......
rm -f fir_fifo
mkfifo fir_fifo
python3 ./sc_plotDFT.py &
cp ../fir/hls_component/syn/verilog/*.dat .
./obj_dir/Vfir

    SystemC 3.0.2-Accellera --- Jun 13 2025 17:49:45
    Copyright (c) 1996-2025 by all Contributors,
    ALL RIGHTS RESERVED
Info: (I703) tracing timescale unit set: 100 ps (sc_fir_tb.vcd)
[   0] y=    0:Y=    0 OK
[   1] y=  916:Y=  916 OK
[   2] y= 3064:Y= 3064 OK
[   3] y= 7281:Y= 7281 OK
[   4] y=12313:Y=12313 OK
[   5] y=17000:Y=17000 OK
[   6] y=20017:Y=20017 OK
    ......
[4795] y=19408:Y=19408 OK
[4796] y=17472:Y=17472 OK
[4797] y=15344:Y=15344 OK
[4798] y=13738:Y=13738 OK
[4799] y=13102:Y=13102 OK
```

고위 합성으로 얻은 fir()의 베릴로그 RTL의 시뮬레이션 파형을 살펴보면 다음과 같다. SHIFTER_LOOP와 MACC_LOOP 구문이 연속적으로 진행된 모습을 관찰 할 수 있다. MACC_LOOP의 경우 초기 준비에 1클럭, 반복 준비에 5 클럭 지연 후 매 클럭 마다 8번의 곱셈과 누산이 연속(pipeline)되는 것을 볼 수 있다.

4. Quartus 및 QuestaSim 설치

Quartus 는 알테라(Altera) FPGA 개발 도구를 모아놓은 패키지다. 자사의 FPGA에 특화된 합성과 배치배선 도구 그리고 QuestaSim HDL 시뮬레이터로 구성되었다. QuestaSim은 지멘스 EDA의 시뮬레이터 제품으로 무료가 아니지만 1년짜리 시험 사용 라이센스를 발급 받을 수 있다. 라이센스는 재발급 할 수 있다. 횟수 제한 없이 자동 발급된다. 라이센스를 발급 받으려면 아래 링크에서 계정을 생성해 둔다. 물론 무료다.

https://licensing.intel.com

아래의 링크에서 설치 파일(리눅스용 표준 판)을 내려받는다.

https://www.intel.com/content/www/us/en/collections/products/fpga/software/downloads.html

Intel® Quartus® Prime Standard Edition Design Software Version 24.1 for Linux

또는,

https://www.intel.com/content/www/us/en/software-kit/849752/intel-quartus-prime-standard-edition-design-software-version-24-1-for-linux.html

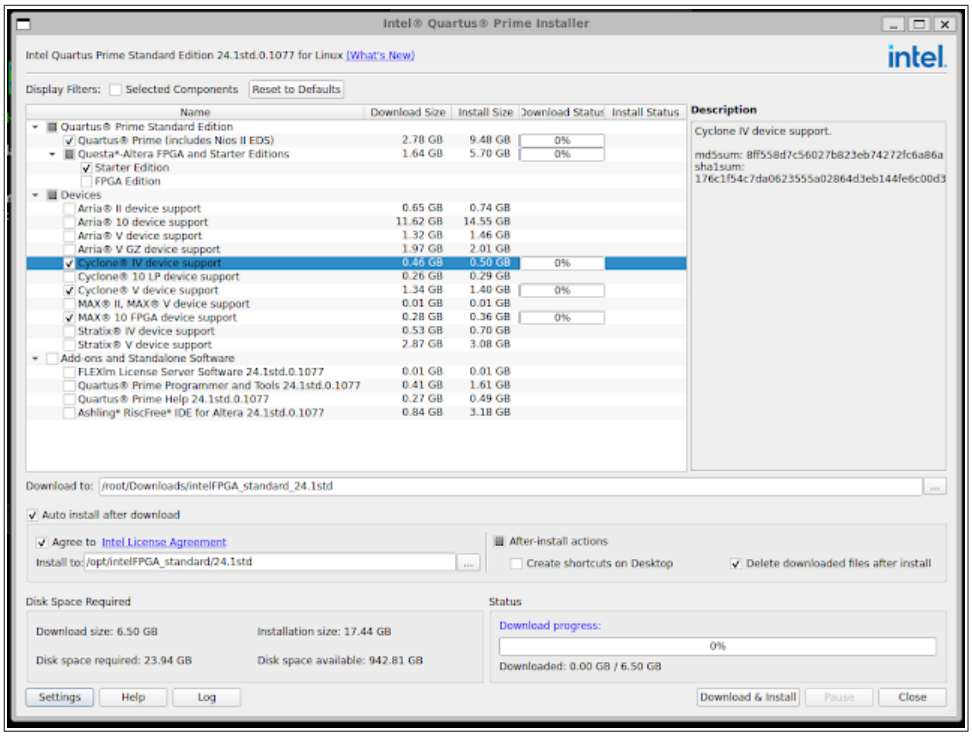
내려받은 파일에 실행 속성이 없을 경우,

```
$ chmod +x qinst-standard-linux-24.1std-1077.run
```

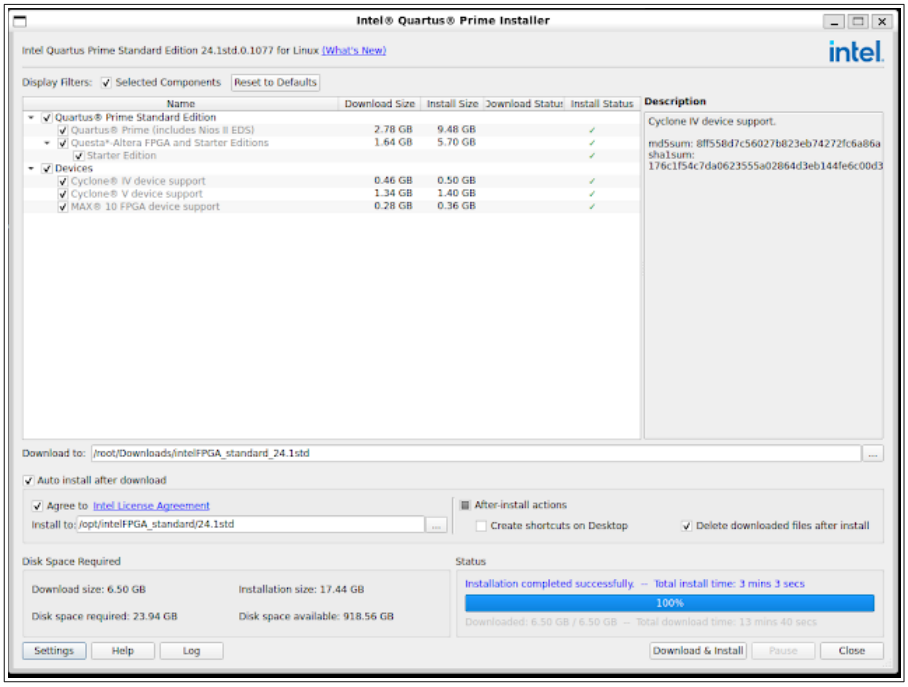
설치 프로그램 실행한다. 도구들의 설치 위치를 편의상 /opt 로 하기 위해 관리자 권한으로 실행 한다.

```
$ cd ~/Downloads
$ sudo ./qinst-standard-linux-24.1std-1077.run
```

설치 프로그램 실행한다. 도구들의 설치 위치를 편의상 /opt 로 하기 위해 관리자 권한으로 실행 한다. 설치 사항으로 Quartus Prime와 Questa-Altera FPGA Starter Edition을 선택한다. FPGA 제품군으로 Cyclone IV 와 V 를 고른다. 설치 위치를 /opt/intelFPGA_standard/24.1std 로 변경한다.

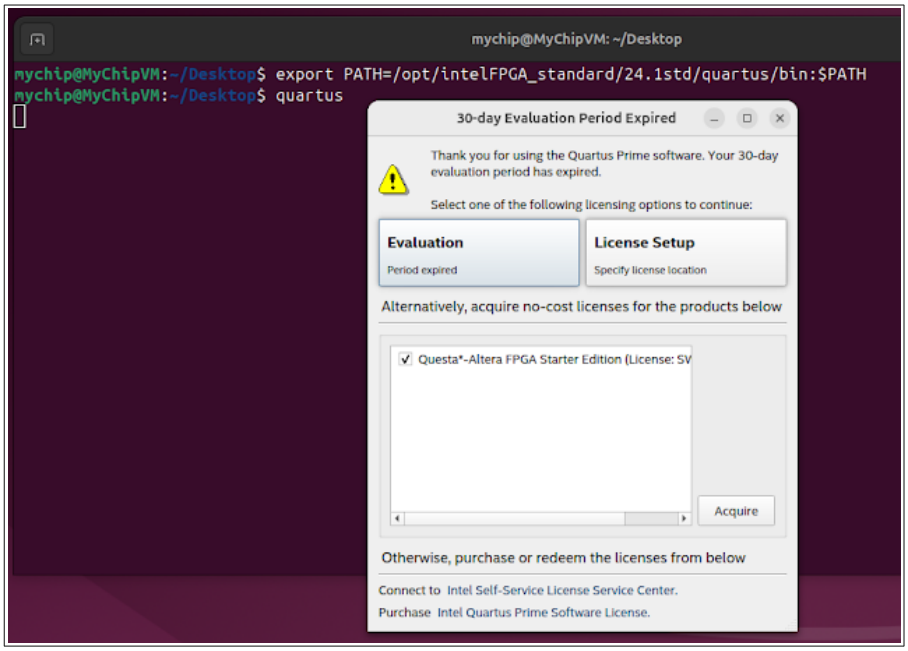


옵션을 확인 하고 설치 시작.



설치를 확인해보자. Quartus의 설치 경로를 환경 변수 PATH에 추가한 후 quartus 실행,

```
$ export PATH=/opt/intelFPGA_standard/24.1std/quartus/bin:$PATH
$ quartus
```



30-일 평가사용(Evaluation) 할 수 있다. 만일 평가기간이 만료되었다고 하면 ~/.altera.quartus 디렉토리를 삭제하면 다시 평가사용 기간이 시작된다.

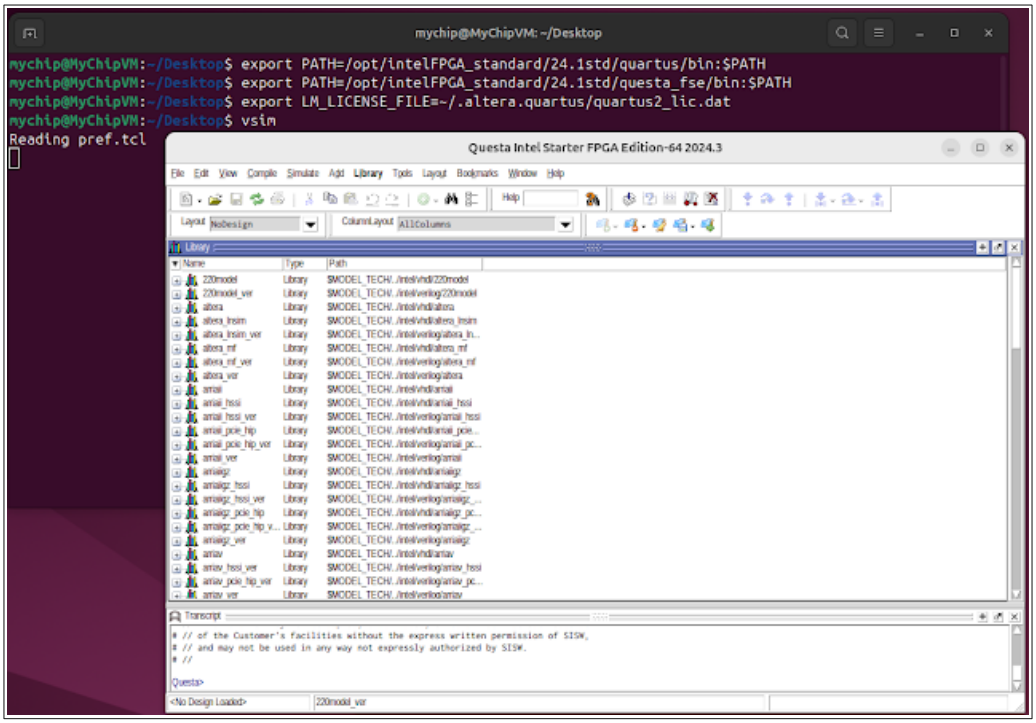
```
$ rm -rf ~/.altera.quartus
```

Questa 는 업계 공인 최고수준의 HDL 시뮬레이터다. Verilog, SystemVerilog, VHDL은 물론 SystemC까지 혼합 시뮬레이션이 가능하다. Intel/Altera에서 이 시뮬레이터의 라이센스를 Quartus 와 함께 무료 공급하고 있다. 라이센스는 "Acquire" 버튼을 눌러 자동으로 발급 받을 수 있다. 라이센스 파일은 ~/.altera.quartus/quartus2_lic.dat 다. 환경 변수에 LM_LICENSE_FILE에 다음과 같이 추가한다.

```
$ export LM_LICENSE_FILE=/home/goodkook/.altera.quartus/quartus2_lic.dat
```

라이센스를 설정하면 Questa HDL 시뮬레이터를 무료로 1년간 사용할 수 있다. 기간이 만료되면 다시 발급 받아 사용할 수 있다. Questa 가 설치된 경로를 추가한 후 실행시켜보자.

```
$ export PATH=/opt/intelFPGA_standard/24.1std/quartus/bin:$PATH
$ export PATH=/opt/intelFPGA_standard/24.1std/questa_fse/bin:$PATH
$ export LM_LICENSE_FILE=~/.altera.quartus/quartus2_lic.dat
$ vsim
```



5. FIR 필터의 "내 칩" 레이아웃

"남의 칩"으로 검증을 마친 FIR 필터를 "내 칩"으로 제작하기 위한 레이아웃을 생성해보자. 작업 디렉토리로 이동,

```
$ ~/ETRI050_DesignKit/Tutorials/2-8_Lab6_FIR8_c_untimed_Vitis-HLS/ETRI050
```

준비된 Makefile 을 사용하기로 한다. 합성과 배치배선을 거쳐 레이아웃을 생성한다.

```
$ make synthesize
```

시뮬레이션.

```
$ cd simulation
$ make build_net
$ make run
```

배치

```
$ ~/ETRI050_DesignKit/Tutorials/2-8_Lab6_FIR8_c_untimed_Vitis-HLS/ETRI050
$ make place
```

배선

```
$ make route
```

표준 셀 병합

```
$ make migrate
```

LVS

```
$ make lvs
```

코어 크기

```
$ make size
```

코어의 크기가 1023.600 x 985.050 다. 아쉽게도 "내 칩" MPW에서 요구하는 면적보다 약간 크다. 설계 최적화가 필요하다. 코어의 GDS(레이아웃)을 보는 과정은 다음과 같다.

```
$ cd layout
$ make mag2gds
$ make klayout
```

6. 맺음말

C 로 기술한 FIR 필터 알고리즘을 고위합성 도구 VitisHLS를 사용하여 베릴로그 RTL로 변환 하고 FPGA로 검증을 실시하였으며 이를 최종적으로 "내 칩"으로 제작하기 위한 레이아웃까지 생성하는 과정을 기술했다. 최고 추상화 수준에서 가장 낮은 레이아웃에 이르기까지 오픈-소스 도구와 무료로 제공되는 FPGA 도구들이 사용됐다. 추상화 수준이 낮춰질 때마다 시뮬레이션과 에뮬레이션 검증을 거쳤다. SystemC로 작성된 테스트벤치가 재사용되고 있다는 점에 주목한다. 반도체 설계 자동화 도구들의 소개를 위한 목적이므로 기법들을 자세히 설명하지 못했다. 깃허브를 통해 배포된 예제를 따라 해보면서 설계과정을 익히기 바란다. 그리고 그 과정이 필요한 이유를 토론해보기 바란다. C/C++ 언어를 이해 한다면 반도체 칩을 설계 할 수 있다는 말이 허사가 아님을 알아챘길 바란다. 각 추상화 단계별 설계 기법의 세밀한 내용은 향후 학습 노트에서 설명할 것이다. 중소규모 산업과 교육 현장에서디 지털 반도체 설계에 도구의 장벽은 더 이상 존재하지 않는다.

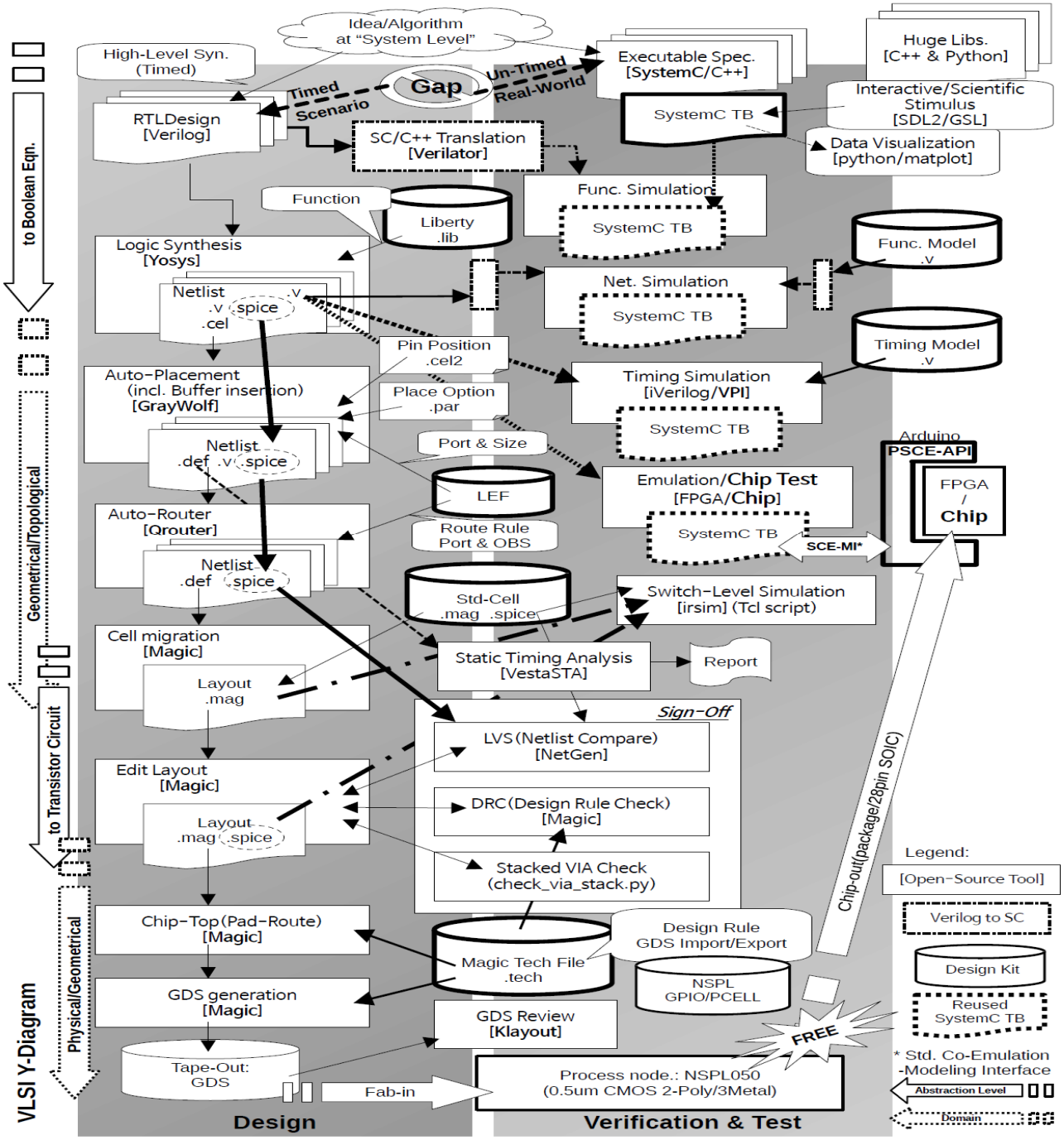
[참고]

[1] "내 칩 제작 서비스 디자인 킷" 배포,
https://github.com/GoodKook/ETRI-0.5um-CMOS-MPW-Std-Cell-DK.git

[2] Roadmap and Recommendations for Open Source EDA in Europe,
https://fossi-foundation.org/resources/eu-roadmap

[3] Are Open-Source EDA Tools Ready for a Multi-Million-Gate, Linux-Booting RV64 SoC Design?,
https://arxiv.org/html/2405.04257v2

[4] OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain,
https://par.nsf.gov/servlets/purl/10171024


Idea/Algorithm at "System Level"
High-Level Syn. (Timed)
Gap
Timed Scenario
Un-Timed Real-World
Executable Spec. [SystemC/C++]
Huge Libs. [C++ & Python]
Interactive/Scientific Stimulus [SDL2/GSL]
SystemC TB
Data Visualization [python/matplot]
RTLDesign [Verilog]
SC/C++ Translation [Verilator]
to Boolean Eqn.
Function
Liberty .lib
Func. Simulation
SystemC TB
Logic Synthesis [Yosys]
Netlist .v .spice .cel
Func. Model .v
Net. Simulation
SystemC TB
Timing Model .v
Pin Position .cel2
Auto-Placement (incl. Buffer insertion) [GrayWolf]
Place Option .par
Timing Simulation [iVerilog/VPI]
SystemC TB
Netlist .def .v .spice
Port & Size
Arduino PSCE-API
FPGA / Chip
LEF
Emulation/Chip Test [FPGA/Chip]
SystemC TB
SCE-MI*
Auto-Router [Qrouter]
Route Rule Port & OBS
Netlist .def .spice
Std-Cell .mag .spice
Switch-Level Simulation [irsim] (Tcl script)
Geometrical/Topological
Cell migration [Magic]
Static Timing Analysis [VestaSTA]
Report
Layout .mag
Sign-Off
LVS (Netlist Compare) [NetGen]
Edit Layout [Magic]
DRC (Design Rule Check) [Magic]
Layout .mag .spice
Stacked VIA Check (check_via_stack.py)
to Transistor Circuit
Chip-out(package/28pin SOIC)
Legend:
[Open-Source Tool]
Chip-Top (Pad-Route) [Magic]
Design Rule GDS Import/Export
Verilog to SC
Magic Tech File .tech
Design Kit
NSPL GPIO/PCELL
Reused SystemC TB
GDS generation [Magic]
GDS Review [Klayout]
FREE
* Std. Co-Emulation -Modeling Interface
Abstraction Level
Domain
Tape-Out: GDS
Fab-in
Process node.: NSPL050 (0.5um CMOS 2-Poly/3Metal)
VLSI Y-Diagram
Physical/Geometrical
Design
Verification & Test