**RFSoC:LOLENC**

QuIQCL RFSoC LOw LatENCy Manual

**- Written by: jhpark snu -**

[alexist@snu.ac.kr](mailto:alexist@snu.ac.kr)



# Contents

[Contents 2](#_Toc156060235)

[1. Introduction to RFSoC 3](#_Toc156060236)

[1.1.RFSoC Chip 3](#_Toc156060237)

[1.2.Ultrascale Plus Architecture 3](#_Toc156060238)

[1.3.ZCU111 4](#_Toc156060239)

[1.4.AXI Bus 6](#_Toc156060240)

[1.5.Real Time Input Output Concept 8](#_Toc156060241)

[1.5.Overview of Current Architecture 8](#_Toc156060242)

[2. PL Side Architecture 9](#_Toc156060243)

[2.1.DAC 9](#_Toc156060244)

[2.3.DAC Controller 10](#_Toc156060245)

[2.4.RTO(B) & GPO 12](#_Toc156060246)

[2.5.TTL 13](#_Toc156060247)

[2.6.TTLx8 13](#_Toc156060248)

[2.7.TimeController 14](#_Toc156060249)

[3. PS Side Firmware 14](#_Toc156060250)

[3.1.Firmware 14](#_Toc156060251)

[3.1.1.Running ELF 14](#_Toc156060252)

[3.1.2.Board Initialization 16](#_Toc156060253)

[3.1.3.Command Set 17](#_Toc156060254)

[4. Experiment Software 18](#_Toc156060255)

[4.1.Compiler 18](#_Toc156060256)

[4.2.PL Modules 18](#_Toc156060257)

[4.2.1.DAC 19](#_Toc156060258)

[4.2.2.TTL 19](#_Toc156060259)

[4.2.3.TTLx8 20](#_Toc156060260)

[4.2.4.TimeController 20](#_Toc156060261)

[5. Project Creation 20](#_Toc156060262)

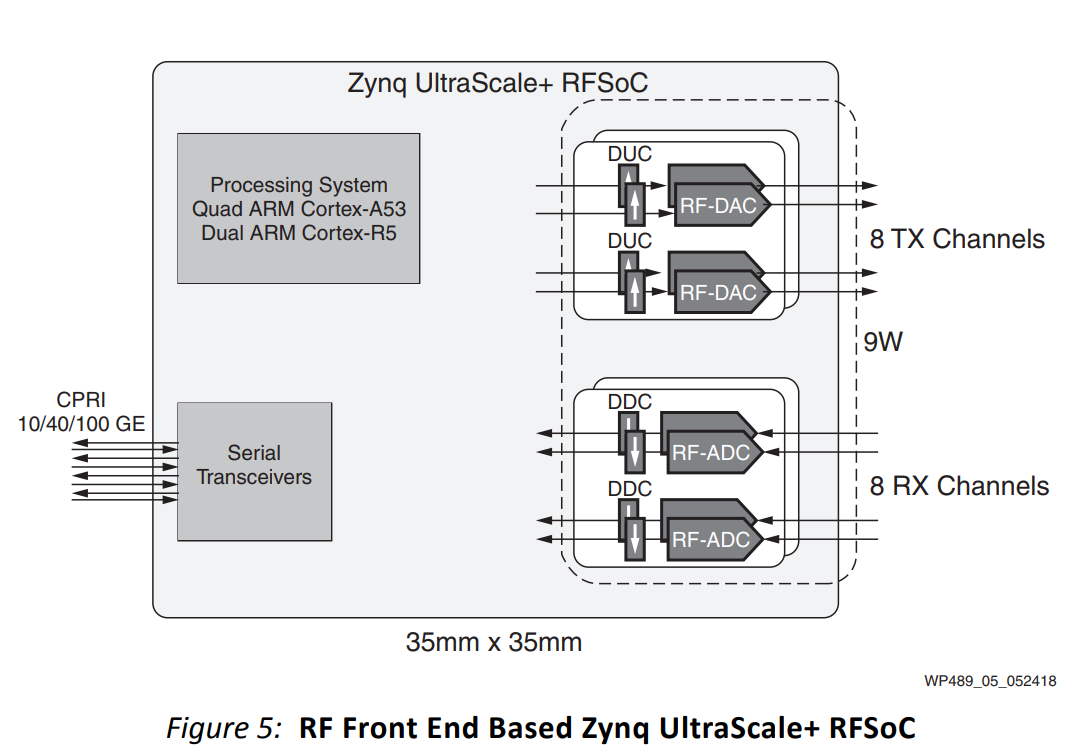
[5.1.Vivado Creation 20](#_Toc156060263)

[5.2.Vitis Creation 20](#_Toc156060264)

# 1. Introduction to RFSoC

## 1.1.RFSoC Chip

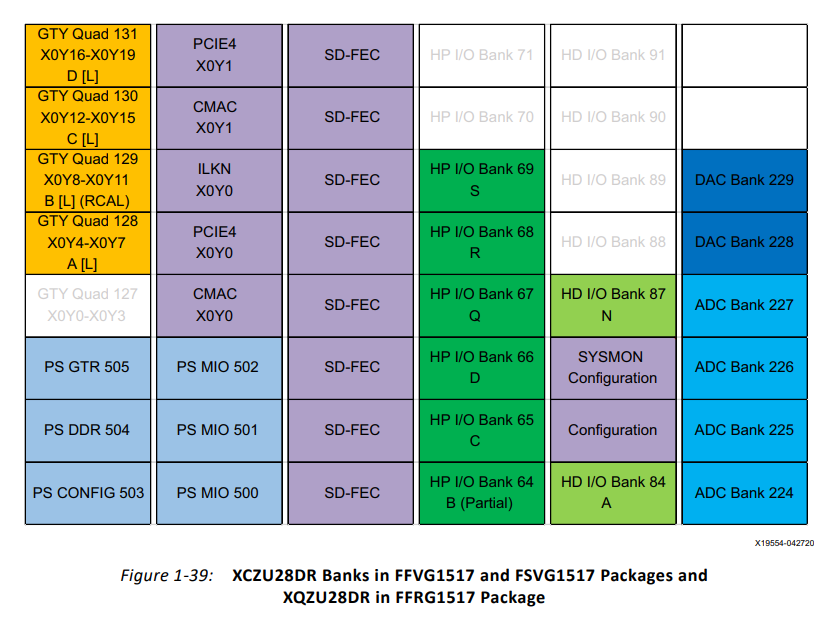
RFSoC는 ARMv8의 CPU및 ADC, DAC가 IC에 내장되어 있는 Xilinx의 Ultrascale+ 계열의 IC에 해당한다. Gen1,2,3은 RFSoC칩의 세대를 나타내며 Gen의 숫자가 증가함에 따라 기타 기능이 추가된다. 본 연구실에서 ZCU111의 경우 XCZU28DR 칩이 내장되어 있고 Gen1에 해당하며 8개의 ADC, 8개의 DAC가 내장되어 있다. 각각 sampling rate는 4GBPS, 6.4GBPS에 해당한다. ADC, DAC의 경우 RF Generator라는 IP로 접근이 가능하며 기존에 IP를 생성하던 방식과 동일한 방식으로 생성이 가능하다.



RFSoC의 경우 AMD의 CPU가 내장되어 있는 hardcore방식에 해당한다. ARTIQ의 경우 OpenRISC시리즈를 기반으로 한 mor1kx를 사용하는 것과 대비된다.[[1]](#footnote-1) CPU의 경우 Application Processing Unit(**APU**), 그리고 Realtime Processing Unit(**RPU**)로 구성된다. XCZU28DR의 경우 4개의 APU(quadcore), 그리고 2개의 RPU로 구성된다. APU의 경우 통상적인 CPU로 생각할 수 있으며 32bit architecture를 사용하는 ARMv7와는 다르게 64bit ARMv8 architecture이다. 이로 인해 기존의 Zynq7000이 사용하던 ARMv7과는 다른 exception level을 사용하는 등 ARMv7과 어느 정도의 차이점이 존재한다. RFSoC에 관한 더 자세한 내용은 해당 문서를 참고할 수 있다..[[2]](#footnote-2)[[3]](#footnote-3)

## 1.2.Ultrascale Plus Architecture

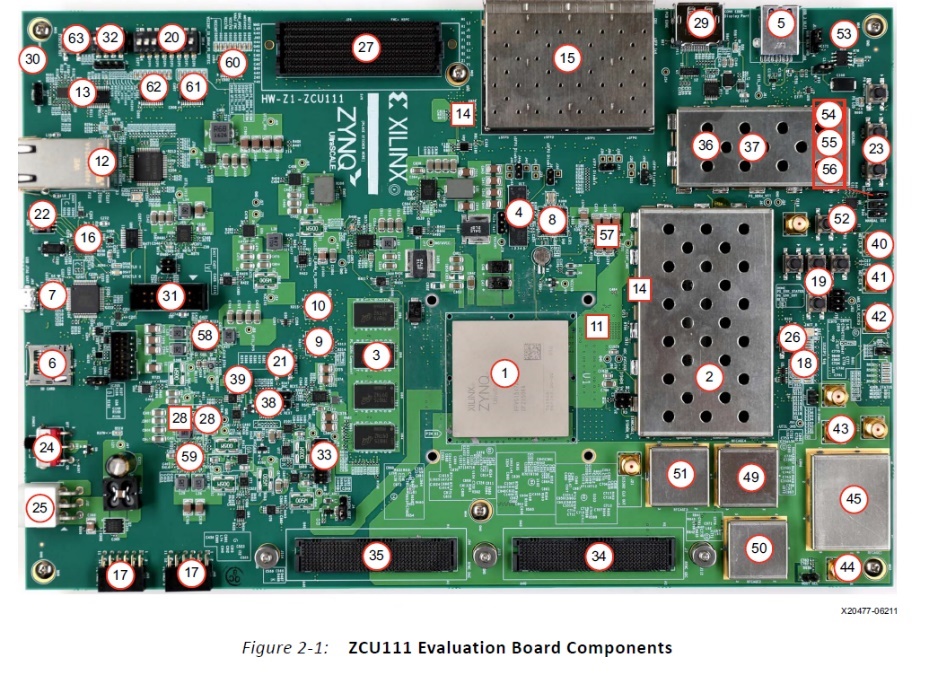
Xilinx의 FPGA에는 Intel의 i9000 시리즈, i10000시리즈와 같이 년도에 따라 발전하며 여러가지 시리즈들이 출시되었다. 7 Series 이후 Ultrascale, Ultrascale Plus 구조가 출시되었으며 공정 두께, 그리고 IP 모듈 등등이 변경되었다. 또한 같은 시리즈 내에서도 여러 버전이 나누어지며 7 Series의 경우 Spartan, Artix, Kintex, Vertex등이 있으며 ZCU111에 존재하는 XCZU28DR의 경우 Ultrascale + RFSoC에 해당한다.



FPGA의 IO들은 bank라는 group으로 묶여 있으며 XCZU28DR의 경우 높은 성능을 보이는 High Performance(**HP**) bank와 비교적 낮은 성능을 보이는High Density(**HD**) bank로 구분되어 있다. HP bank의 경우 OSERDES IP를 이용하여 최대 8배 빠른 시그널을 만들어 낼 수 있다.

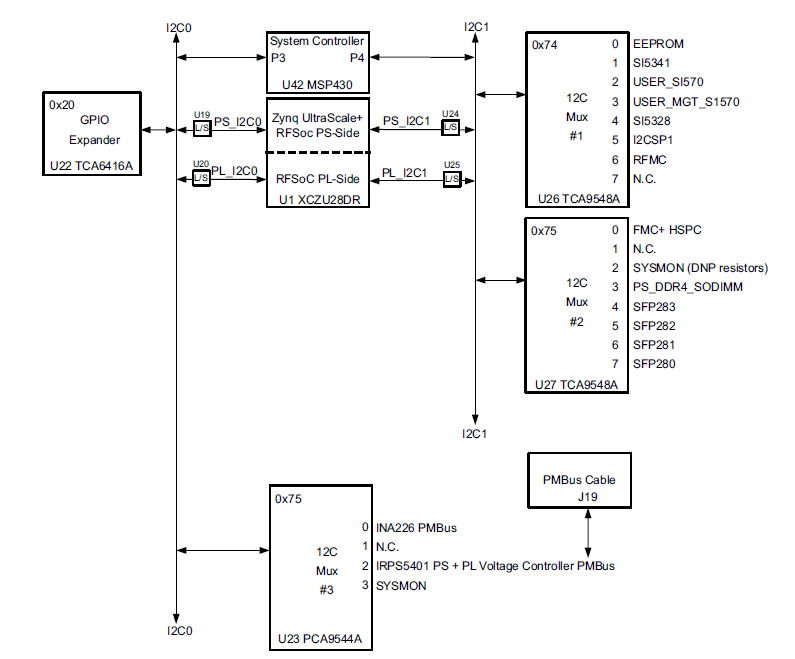
## 1.3.ZCU111

ZCU111에는 RFMC, 그리고 FMC커넥터가 존재한다. 아래 그림의 34,35번 부품이 RFMC, 그리고 27번 부품이 FMC(VITA57.4)에 해당한다.

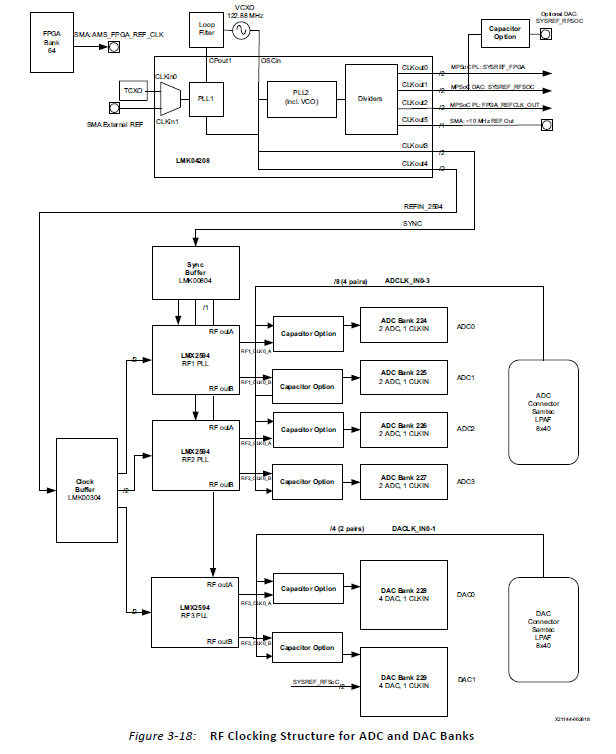


RF신호와 느린 TTL의 경우는 RFMC, 빠른 TTL 신호의 경우 FMC으로부터 입출력되게 된다. 현재 설계 기준으로 1ns 단위의 resolution을 가지고 control이 가능하다. 그외의 TTL신호의 경우 PMOD로부터도 출력이 가능하나 high impedance가 연결되어 있어 빠른 속도의 신호 전송이 요구되는 경우 사용이 불가능하다. RFMC의 경우에도 DAC IO, ADC IO핀을 통해 TTL신호 입출력이 가능하나 HR bank로 연결되어 있어 SERDES와 같은 기능 사용이 제한되어 낮은 timescale의 resolution을 가지게 된다.

보드에는 아래그림과 같이 I2C통신을 위한 칩들이 존재하며 이를 컨트롤하는 코드들은 Xilinx에서 제공되어 있다. 또한 보드 자체에 oscillator및 PLL이 내장되어 있어 RF, 그리고 디지털 클락소스를 만들 수 있다. 내장된 oscillator가 아닌 외부의 클락을 이용할 수도 있으며 이 경우 PLL의 설정을 ZCU111보드 내에서 I2C 통신을 통해 변경해야 한다. 또한 RF 클락의 주파수를 변경하는 경우에도 마찬가지의 작업을 해야 한다.



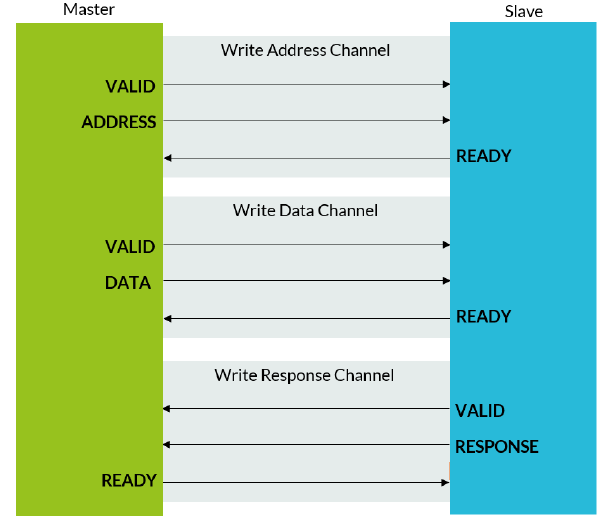
RF클락의 경우 아래와 같은 회로에서 출력이 된다. LMK04028와 LMX2594은 PLL, LMK00304는 클락 버퍼에 해당한다. Xilinx에서 제공하는 펌웨어에서는 LMK04028을 122.88MHz의 oscillator를 입력받아 다시 122.88MHz의 클락을 출력하고 LMX2594에서 곱해진 주파수의 클락을 출력한다. TICS Pro 프로그램[[4]](#footnote-4)을 이용해 GUI상에서 주어진 주파수의 시리얼 통신 데이터를 만들 수 있으며 주어진 시리얼 통신 데이터는 I2C 통신을 이용해 입력이 가능하다.[[5]](#footnote-5)



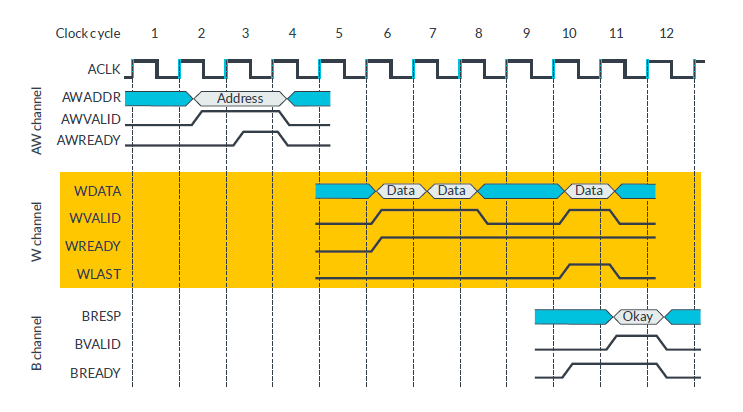
보드와 PC사이에서 이더넷 통신 및 UART통신이 모두 가능하다. 현재 FPGA를 프로그래밍하는 경우와 xil\_printf함수를 이용해 FPGA로부터 PC로 출력하는 경우 시리얼 통신을 수행하며 명령어를 입력, 프로그램 전송 및 실험 데이터 송수신은 이더넷을 이용한다.

## 1.4.AXI Bus

Zynq sereis에서 APU, 그리고 RPU를 통틀어 Processing System(**PS**), 그리고 이를 제외한 부분을 Programmable Logic(**PL**)이라고 지칭한다. PS와 PL사이에서는 Advanced eXtensible Interface(**AXI**) 버스를 통해 통신한다. AXI bus는 아래 그림처럼 handshake 방식의 통신 protocol이다. Master로부터 slave로 메모리 및 IO 주소를 전달하는 write address channel(**WA**), 그리고 slave로 데이터를 전송하는 write data channel(**W**), 마지막으로 데이터를 전송 결과를 전송하는 write response channel(**B**) 버스들로 구분된다.



각각의 channel은 현재의 데이터가 전송하고자 하는 데이터인지 나타내는 valid signal, 그리고 이러한 데이터를 받을 수 있는지 상태를 나타내는 ready signal이 존재한다. Valid, 그리고 ready signal이 모두 high인 경우에만 데이터 전송이 발생한다. 아래는 데이터 전송 과정 예시를 보여준다. 이외에도 어느 부분의 data가 전송되는지 나타내는 strobe signal, 얼마의 데이터가 전송되는지 나타내는 len signal, 그리고 현재의 데이터가 마지막 데이터 전송임을 나타내는 last signal이 존재한다. 그외에도 qos, user와 같은 signal이 존재하며 해당 사항은 아래의 문서에서 자세히 알 수 있다.[[6]](#footnote-6)

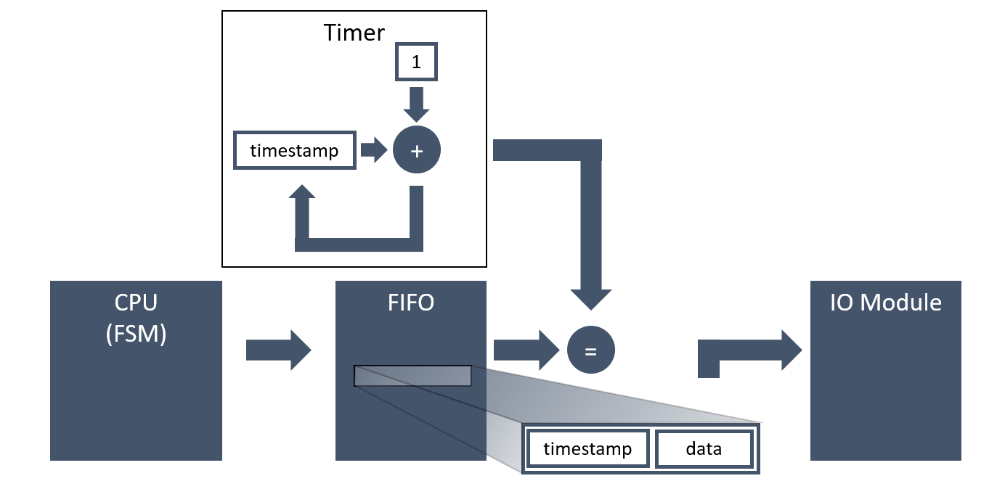


현재 RFSoC Verilog 코드는 모두 AXI bus에 기반하여 작성되어 있으며 CPU는 memory mapped IO 방식을 사용한다. 따라서 특정 모듈에 접근하기 위해서는 해당 모듈의 AXI bus의 주소에 접근하면 된다. 예를 들어 AXI bus의 주소가 0x001000으로 지정된 경우 아래와 같은 코드를 통해 데이터를 전송할 수 있다. 이 때 volatile이라는 키워드를 추가하였는데 0x001000이라는 주소에 데이터를 입력하는 코드를 컴파일러에서 최적화하여 캐쉬에만 write하는 경우를 방지하기 위한 키워드이다.

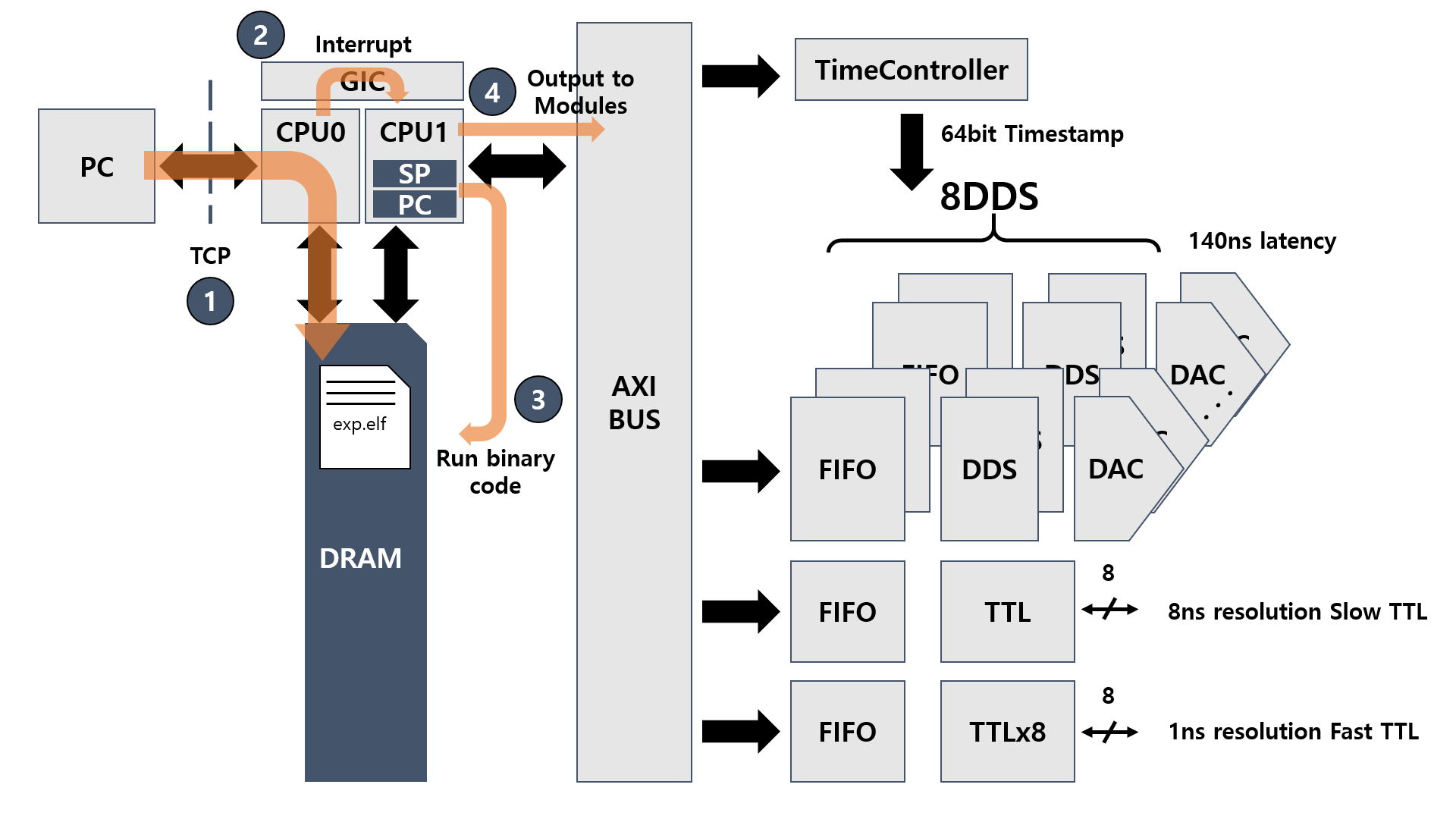
|  |
| --- |
| 01 int64\_t data;  02 volatile int64\_t \* axi\_addr = (volatile int64\_t \*) 0x001000;  03 \*(axi\_addr) = data; |

## 1.5.Real Time Input Output Concept

Real Time Input Output(**RTIO**)는 실시간으로 정확한 타이밍에 신호를 내보내거나 받아들이는 것이다. 예를 들어 간략화 된 RTIO 시스템은 아래 그림과 같다. 현재 RFSoC시스템의 경우 128bit의 First In First Out(**FIFO**) 구조에 데이터들이 저장되어 있으며 각각의 데이터는 instruction이라고 칭한다. 이러한 instruction이 나가는 정확한 시간인 64bit timestamp와 해당 시간에 나가는 데이터인 64bit data로 구성된다. Timer에서 timestamp를 매번 클락마다 1씩 증가시키며 FIFO의 top에 해당하는 instruction의 timestamp와 일치하는 경우 output으로 내보내게 된다.



## 1.5.Overview of Current Architecture



현재 RFSoC의 간략한 구조는 다음과 같다. 현재는 4개의 APU 중 두 개의 APU만을 구동하고 있다. 첫번째 CPU는 TCP통신을 하고 PC와 직접적인 command를 받아 CPU에서 함수를 실행하거나, 실험 코드를 DRAM에 저장한다. 두번째 CPU의 경우 첫번째 CPU로부터 인터럽트 신호를 받아 DRAM에 저장되어 있는 코드를 실행한다. 이 때 General Interrupt Controller(**GIC**)는 CPU에 연결된 인터럽트 신호들을 모두 관리한다. CPU와 같은 부분들을 Processing System(**PS**)라고 칭한다.

반대로 Verilog Code로 작성된 코드가 작동하는 FPGA 구역을 Programming Logic(**PL**)이라고 칭한다. 모든 real time input, output은 현재 FPGA내부의 timestamp에 맞추어 output을 하게 된다. TimeController는 FPGA의 클락마다 timestamp를 1씩 증가시키고, CPU로부터 명령어를 받아 timestamp를 초기화하거나, 시작, 정지를 하게 된다.

FIFO는 AXI bus로부터 데이터를 입력받을 수 있으며 해당 데이터를 Xilinx의 IP인 fifo\_generator에 저장하게 된다. 총 128bit 데이터로 64bit은 timestamp, 64bit은 data에 해당한다. Digital Analog Convertor(**DAC**)는 RFSoC FPGA칩 내부에 내장된 회로에 해당한다. Direct Digital Synthesizer(**DDS**)는 Verilog code로 작성된 코드이며 FIFO로부터 받은 instruction에 기반하여 디지털 sin파를 만들게 된다.

TTL의 경우 디지털 신호를 output하는 모듈이며 8ns의 resolution을 가진 TTL 모듈과 1ns의 resolution을 가진 TTLx8모듈이 존재한다. 각각의 모듈에 8개 핀들이 존재하며 각 모듈에서 동일한 시간을 내보낼 경우 해당 신호는 무시되므로 주의해야 한다.

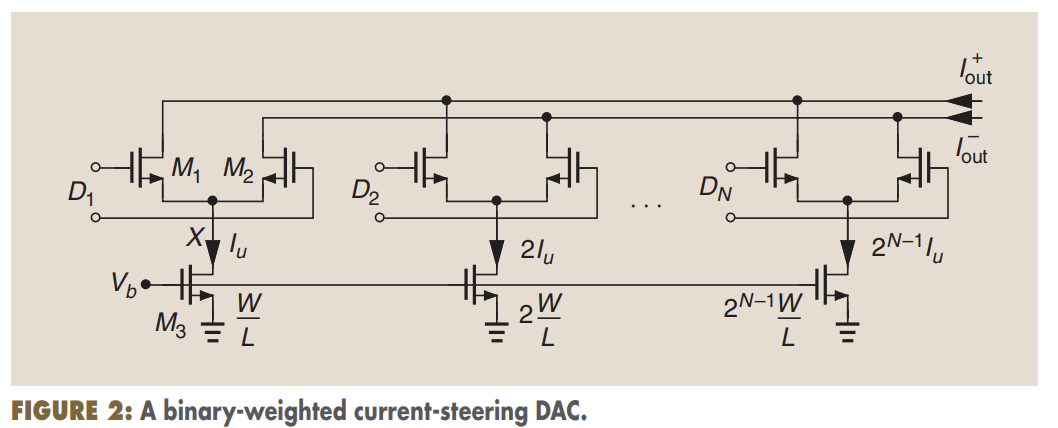
작동 과정은 아래와 같다.

1. PC로부터 TCP를 통해 RFSoC의 DRAM에 ELF 파일 저장
2. 코드 저장 후 다른 CPU로 인터럽트 신호 전송
3. 인터럽트를 받은 CPU에서 코드 실행
4. 주어진 코드에서 각각의 module로 instruction 전송

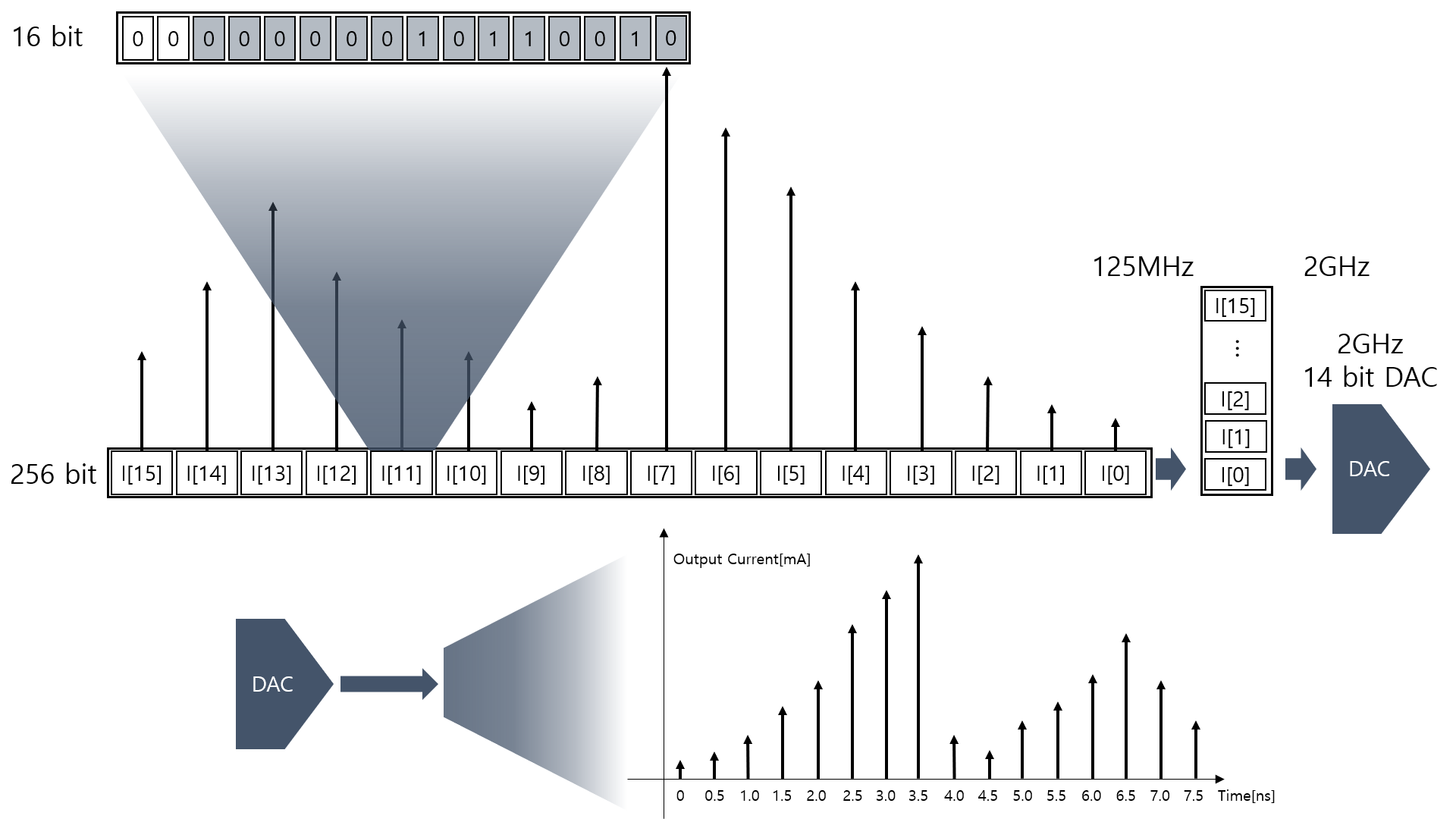
# 2. PL Side Architecture

## 2.1.DAC

RFSoC의 DAC는 대부분의 DAC에서 사용하는 current steering 방식의 DAC를 사용한다.[[7]](#footnote-7) Current steering이란 아래 그림과 같이 하나의 전류 source를 이용해 두개의 트랜지스터에 서로 다른 전압을 인가하여 differential 신호를 발생시키는 방식이다.[[8]](#footnote-8) 여기서 Dj는 DAC output에 해당하는 I = [D13, D12, …,D0]중 j번째 디지털 신호 bit 에 해당한다.



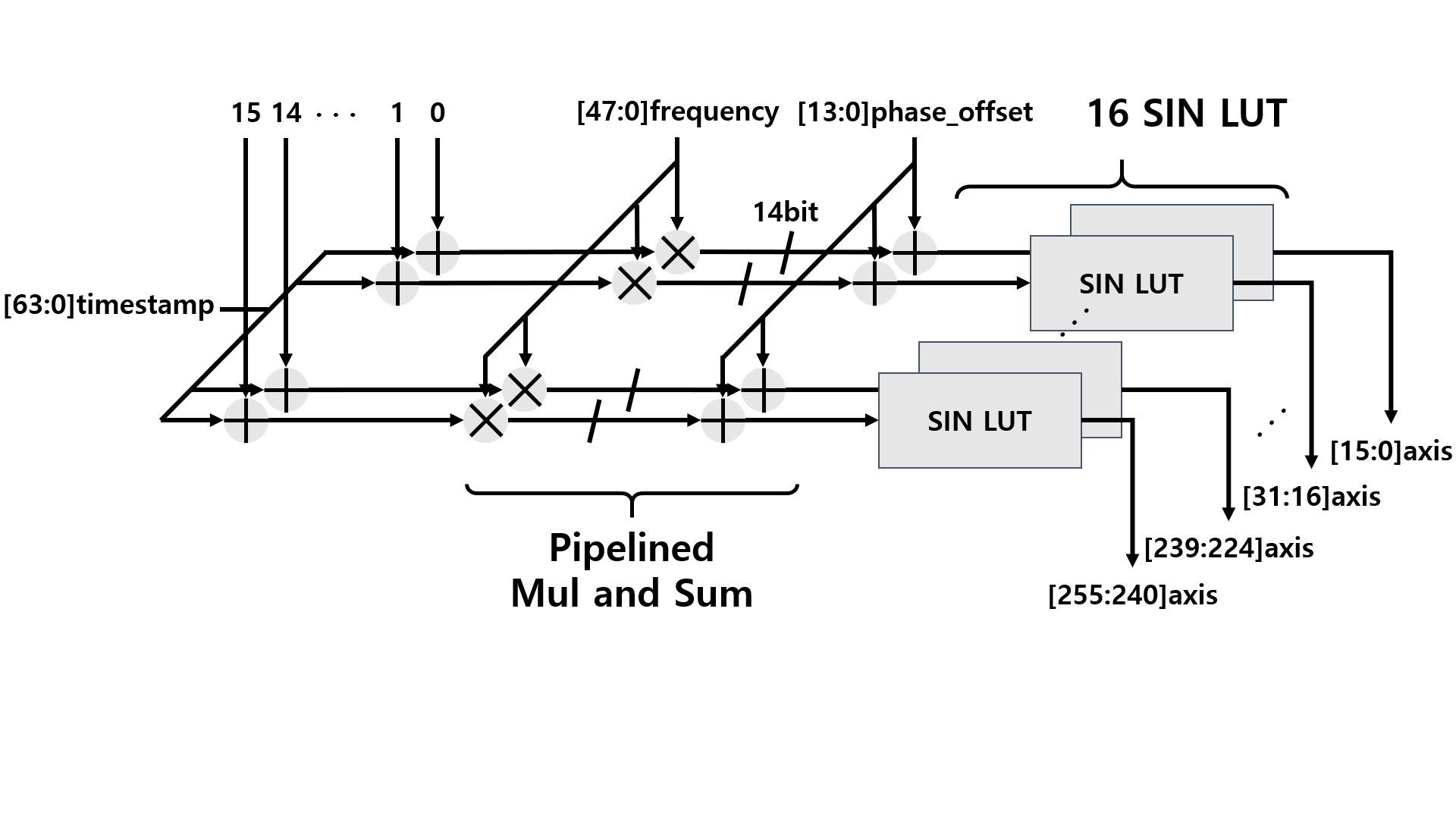
RFSoC는 14bit DAC에 해당하며 LSB부터 MSB까지 이러한 디지털 신호들을 시간순으로 배치하여DAC에 입력하여 순서대로 output을 내보낸다. 구체적으로 현재 RFSoC 구성에서는 256bit의 데이터를 FPGA 매 클락마다 공급하게 된다. FPGA는 125MHz, DAC는 2GHz의 주파수에서 작동하고 있다. 이때 256bit의 데이터에서 [15:0] bit는 0ns에 출력되는 output, [31:16] bit는 0.5ns에 출력되는 output, [47:32] bit는 1.0ns에 출력되는 output에 해당하여 [255:240] bit의 경우 7.5ns에 출력되는 output에 해당한다. 단, 이 때 RFSoC의 DAC는 2’s complement를 사용하여 16bit의 output에서 MSB 2bit의 경우 output의 부호에 해당된다.



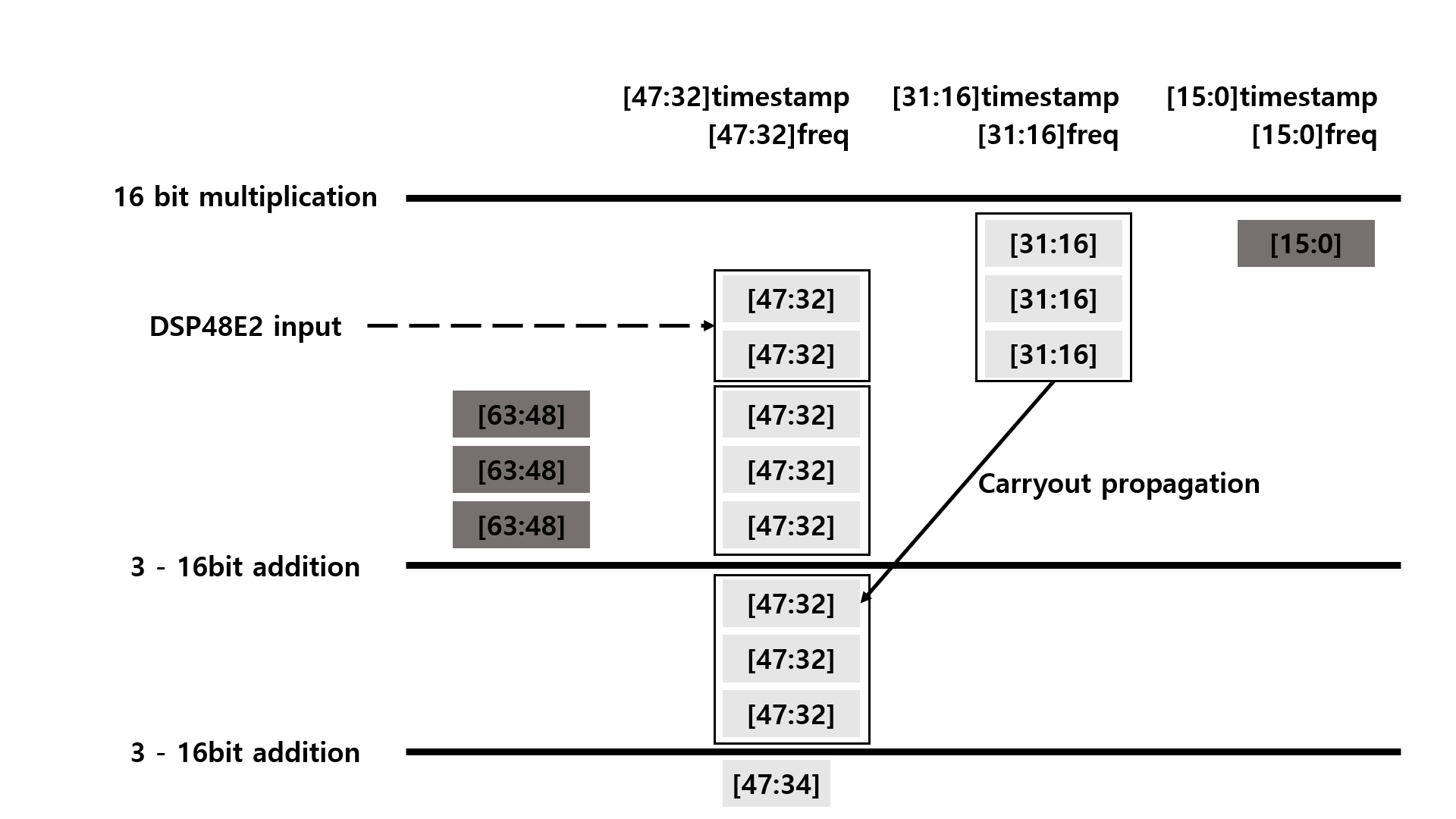
## 2.3.DAC Controller

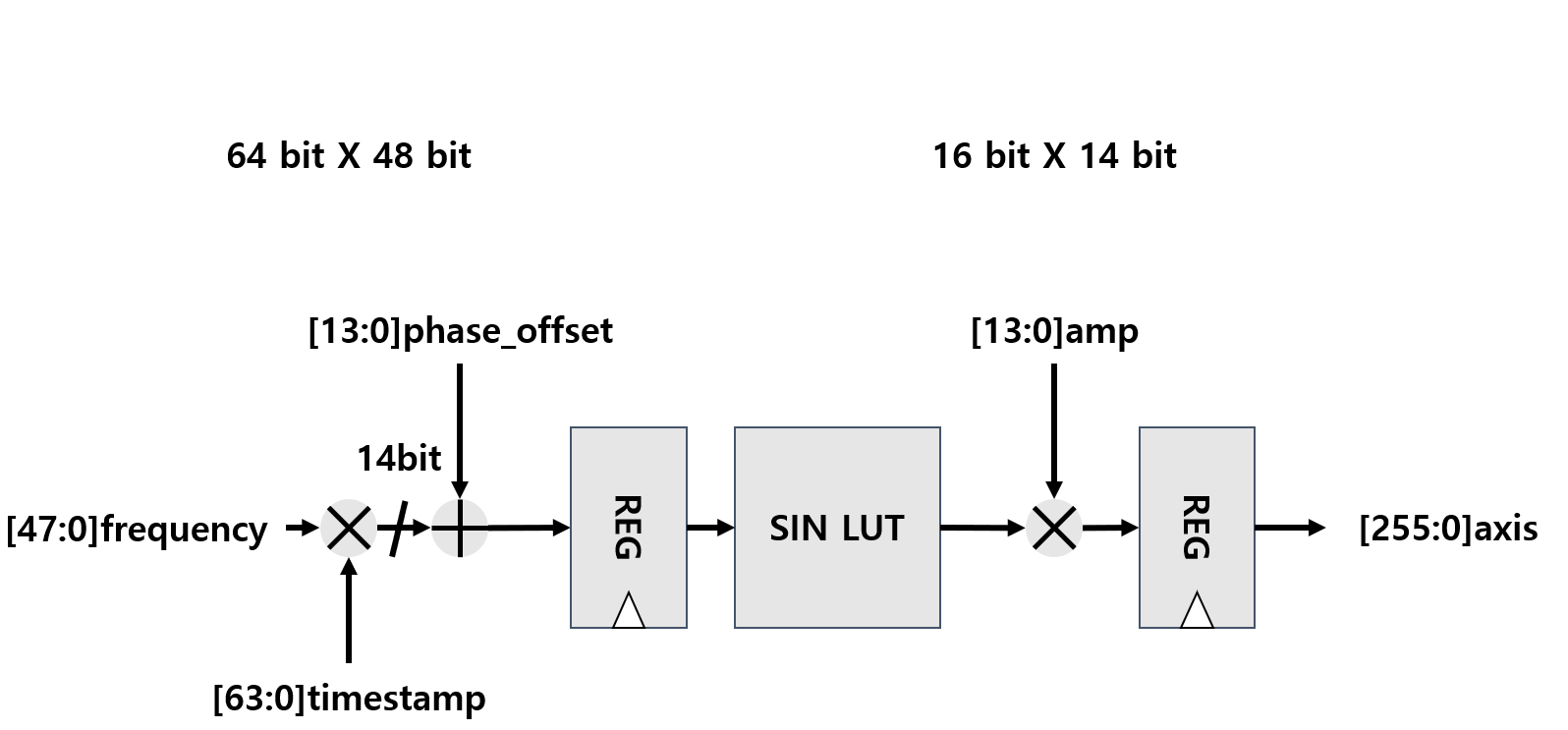
DAC Controller는 RFSoC의 DAC에 입력될 디지털 Sin파를 생성하고 입력하는 모듈이다. 아래의 식과 같이 주어진 48bit의 주파수(), 14bit의 진폭(), 14bit의 위상()에 해당하는 sin파를 만들어낸다. 이 때 시간 t는 timestamp controller에서 만들어내는 timestamp에 해당하는 값을 이용한다. 따라서 주파수를 바꾸는 경우 가 급격하게 변화하여 output의 위상 또한 급격하게 변화한다.

DAC Controller로부터 시간이 증가하는 순으로 16개의 sin output을 만들기 위해 아래 그림과 같이 parallel한 구조를 가지고 있다. TimeController 모듈로부터 공급받은 timestamp값을 DSP IP로 구성된 모듈을 통해 주파수와 곱한 뒤 위상값을 더해주게 된다. 이후에는 해당 위상값을 Xilinx에서 제공하는 Look Up Table(**LUT**)에 입력하여 sin output을 얻게 된다.



이 때 negative slack을 방지하기 위해 아래와 같이 DSP를 pipelining하여 구현하였다.





DAC Controller에는 64bit의 데이터가 입력되며 MSB 4 bit는 입력 데이터가 어떤 데이터에 해당하는지 구분하는 데이터이며 각각의 경우 아래에 해당하는 값을 가진다. DAC Controller는 해당 instruction들을 RTO 모듈을 통해 입력 받는다.

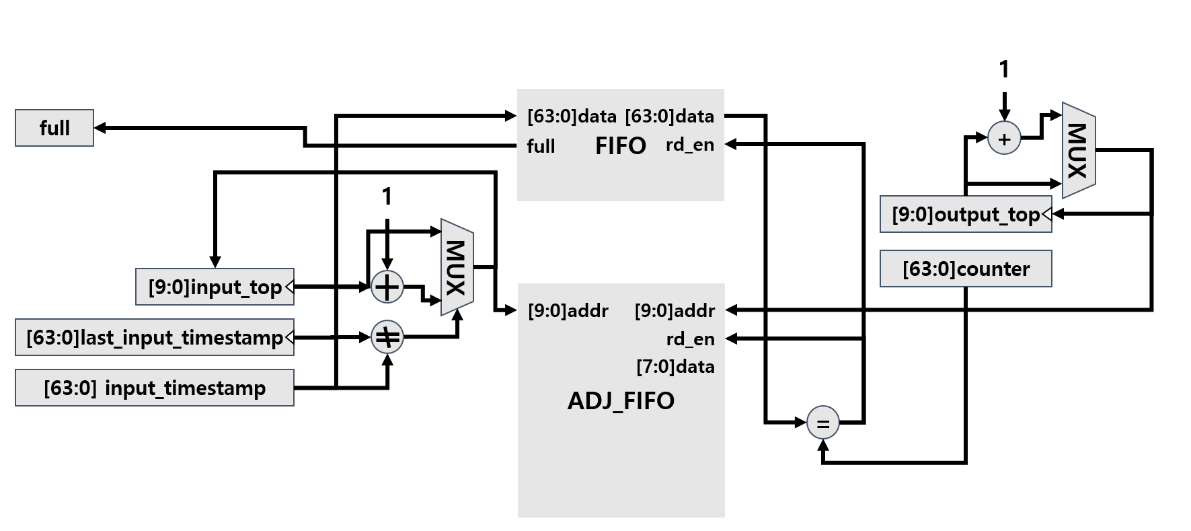
|  |
| --- |
|  |

## 2.4.RTO(B) & GPO

Real Time Output(RTO) 모듈은 AXI bus를 통해 입력된 instruction을 FIFO에 저장하고 정해진 timestamp에 output하게 된다. 이때 정상적인 instruction인 경우 selected pin에 High 신호를 output하게 되고 오류가 있는 경우 Low 신호를 output하게 된다. RTO 모듈에서는 여러가지의 오류 상태를 검출하는데 각각의 오류는 아래와 같다. 현재는 오류가 발생하는 경우 이를 무시하고 계속해서 instruction을 출력한다. 아직 오류가 발생했을 때 이 값을 기록하거나 CPU에 알리는 기능(e.g. 인터럽트)은 구현되지 않았다.

|  |
| --- |
| 1. Timestamp Error  출력되는 Timestamp가 현재 timestamp보다 값이 낮은 경우  2. Overflow Error  Fifo의 capacity보다 더 많은 데이터가 입력된 경우 |

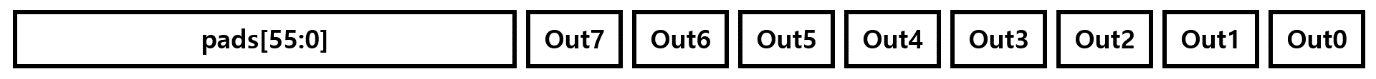
RTO 모듈은 RTOB 모듈과 일반적인 RTO모듈이 존재한다. RTOB 모듈은 입력받은 instruction이 FIFO의 top에 존재하는 instruction의 timestamp와 동일한 timestamp를 가지는 경우 해당 instruction을 override하게 된다. 하지만 다른 timetsamp를 가지는 경우 해당 instruction이 top으로 입력된다. 아래의 그림은 RTOB 모듈의 다이어그램이다. FPGA의 resource를 효율적으로 활용하기 위해 timestamp는 fifo\_generator IP에 해당하며 top instruction이 수정가능한 ADJ\_FIFO는 FPGA의 레지스터를 이용해 구현하였다. DAC\_Controller는 RTO모듈을 이용하므로 FIFO의 top에 있는 instruction을 수정할 수 없다. 하지만 TTL과 TTLx8 모듈의 경우 RTOB 모듈을 이용하므로 FIFO의 top에 있는 instruction을 수정할 수 있으며 이를 이용해 8개의 pin을 control한다.



General Purpose Output(GPO) 모듈은 output된 instruction이 도달했을 때 selected pin이 high인 경우에만 최종 output 모듈로 output하게 된다. Override의 경우 RTO로부터 도달하는 데이터는 무시하고 데이터를 override하게 되는데 현재는 override핀에 데이터를 입력하는 기능이 구현되지 않아 사용할 수 없는 기능이다.

## 2.5.TTL

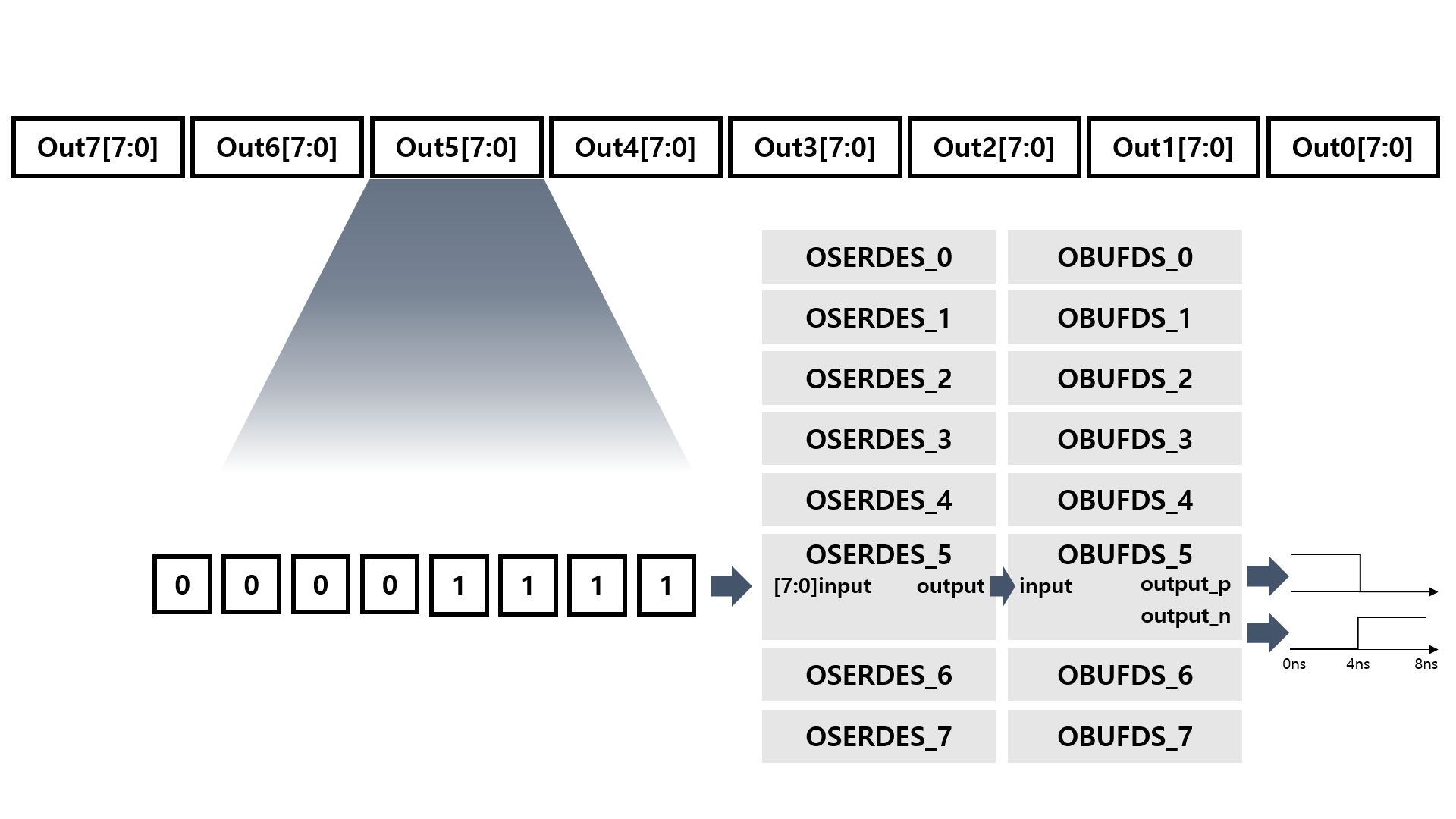
TTL 모듈은 간단한 digital 신호를 output하는 모듈이다. 8개의 핀들을 각각의 TTL 모듈에서 control한다. RTOB모듈에서 입력 받은 instruction에서 LSB 8bit을 각각의 지정된 핀들에 output하게 된다. TTL 모듈의 경우 FPGA의 HD bank에 연결되어 있으므로 최소 8ns resolution의 디지털 신호를 output 할 수 있다



## 2.6.TTLx8

TTLx8 모듈은 FPGA의 HP bank의 IO Select IP 중 OSERDES, OBUFDS IP를 이용하여 1ns resolution을 가지는 Low Voltage Differential Signal (**LVDS**)신호를 생성한다. OSERDES IP는 8개의 비트를 입력받아 8배 빠른 속도로 해당 비트들을 output하게 된다. 예를들어 00001111의 데이터가 입력되는 경우, 0ns : 1, 1ns : 1, 2ns : 1 : 3ns : 1: 4ns : 0, 5ns : 0, 6ns : 0, 7ns : 0이 출력되게 된다. OBUFDS의 경우 single ended signal을 differential signal로 변환해주게 된다. 아직 출력 기능만이 구현되어 있는 상태이지만 verilog코드 상으로 input impedance 를 100Ohm으로 설정 가능하다.

TTL모듈과 마찬가지로 top에 존재하는 instruction을 수정 가능한 RTOB를 사용한다. Differential signal이므로 각 output에 2개의 핀이 사용되어 총 16개의 핀들이 연결되어 있다. 각각의 output은 아래와 같이 8bit 단위로 입력되게 된다. 예를 들어 아래 그림과 같이 5번째 output에 4ns에 High 신호에서 Low신호로 바뀌는 output을 만드는 경우 5번째 output에 00001111를 입력한 후 출력하면 아래와 같이 differential signal로 출력되게 된다.



## 2.7.TimeController

TimeController 모듈은 FPGA의 매 클락(125MHz)마다 timestamp에 1씩 더한 후 해당 timestamp를 다른 모듈들로 output하는 모듈이다. Timeoffset을 지정하여 시작하는 timestamp를 지정하거나, 초기화할 수 있다. 아직 트리거 신호를 입력받아 timestamp 증가를 시작하는 기능은 구현되지 않았다.

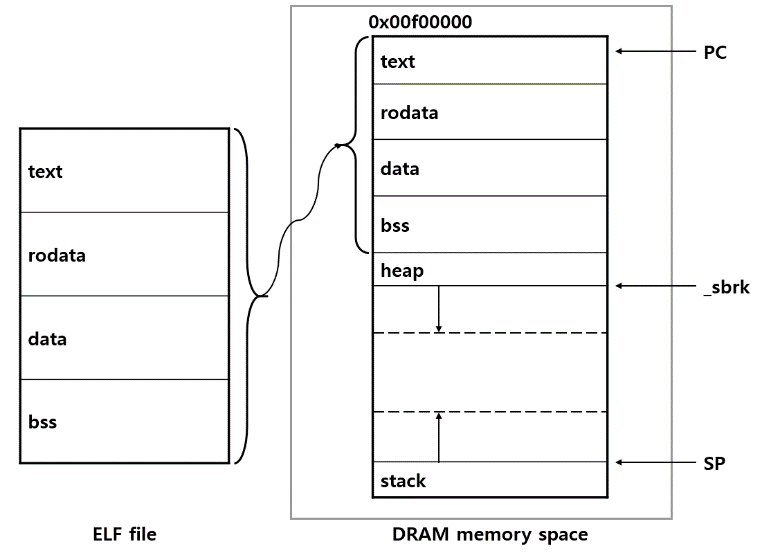
# 3. PS Side Firmware

## 3.1.Firmware

### 3.1.1.Running ELF

CPU는 메모리에 저장되어 있는 코드를 한줄씩 읽어온 뒤 해당하는 코드를 실행하고 결과값을 CPU의 레지스터, 혹은 메모리에 저장하게 된다. 이 때 코드의 메모리 위치를 가리키는 주소를 Pogram Counter(**PC**)라고 하며 메모리에 저장되는 위치를 나타내는 주소는 두가지가 있는데 각각 Stack Pointer(**SP**), 그리고 Heap Pointer가 존재한다. CPU에서는 코드를 실행하기 위해서는 주어진 binary code, 현재 RFSoC의 경우 Executable Loadable File(**ELF**)의 코드 시작 위치로 PC를 이동시켜야 한다. 이러한 코드 시작 위치를 entry point라고 한다.

실험을 위해 작성된 코드를 컴파일하기 위해 gnu-aarch64-none-elf-g++ 컴파일러를 이용하여 ELF로 컴파일 한 후 FPGA(RFSoC)에 데이터를 전송하여 DRAM의 0x800f00000부터 저장한다. 이후 다른 CPU에 인터럽트를 걸어 PC(Program Counter)가 해당 ELF의 entry point로 점프하여 프로그램을 실행하게 된다. 이 때 entry point는 ELF가 실행되기 위한 초기화 과정을 포함한 binary code의 시작점에 해당하며 main의 주소와 다름에 유의한다. ELF의 구조는 아래와 같다.



현재 메모리 관리는 XIlinx에서 주어진 \_sbrk.c 파일에 주어진 코드를 이용해 관리하고 있다. 이 때 \_sbrk함수에서 heap\_pointer를 \_heap\_pointer라는 변수를 이용해 초기화하는데 이는 linker.ld에 지정되어 있는 변수에 해당한다. 따라서 linker.ld에서 해당 변수명을 변경하는 경우 heap pointer가 정확하게 초기화되지 않아 malloc과 같은 메모리 함수를 사용할 수 없게 됨에 주의해야 한다.

앞서 말한 초기화 과정은 BSS영역에 저장된 데이터들에 해당하고 0으로 초기화 되어야한다. 예를 들어 static으로 선언되고 초기화 되지 않은 변수들에 해당한다. 이러한 과정을 위해 컴파일에 startup.S파일을 포함하고 linker 파일 linker.ld에서 entry point를 \_start\_custom\_으로 지정하였다. startup.S는 BSS 영역에 해당하는 변수들을 모두 0으로 초기화한 후 main함수로 jump한다.

ELF를 실행한 후 원래의 함수로 return하기 위해 기존의 레지스터에 존재하던 모든 값들을 stack에 저장한다. sub sp, sp, #256은 스택 포인터의 값을 빼 레지스터 값들을 저장할 공간을 확보한다. str x0, [sp, #8]은 스택으로부터 첫번째 위치에 레지스터 x0 값을 저장하며 x30까지 반복한다. 또한 ELF의 stack pointer또한 기존과 다른 값을 가지기 때문에 컴파일 후 주어지는 stack pointer값을 레지스터에 load 한다. 이후에는 entry point로 jump하게 된다. 단, 기존의 data cache, intstruction cache에 존재하던 값이 사용되는 것을 방지 하기위해 xilinx에서 주어진 cache flush 함수를 실행한 후 해당 작업을 수행한다. 해당 과정에 대한 코드는 run\_binary.c에서 수행하며 아래에 주어져 있다. blr x1코드에서 ELF파일에 주어진 entry point로 jump하게 된다.

|  |
| --- |
| 01 Xil\_DCacheFlush();  02 Xil\_ICacheInvalidate();  03 \_\_asm\_\_ \_\_volatile\_\_ (  04 "sub sp, sp, #256\n\t"  05 "str x0, [sp, #8]\n\t"  06 …  07 "str x30, [sp, #248]\n\t"  08 /\*save SP\*/  09 "mov x0, sp\n\t"  10 "mov x2, #0x700000\n\t"  11 "str x0, [x2]\n\t"  12 /\*save END\*/  13 "ldr x1, [%0]\n\t" // Load sp value to x1  14 "mov sp, x1\n\t" // Move sp value to stack pointer (sp)  15 "ldr x1, [%1]\n\t" // Load main address to x1  16 /\*####JUMP TO MAIN####\*/  17 "blr x1\n\t" // Branch to the address in x1 (MAIN function)  18 /\*return SP\*/  19 "mov x2, #0x700000\n\t"  20 "ldr x0, [x2]\n\t"  21 "mov sp, x0\n\t"  22 /\*return END\*/  23 "ldr x0, [sp, #8]\n\t"  24 "ldr x1, [sp, #16]\n\t"  25 …  26 "ldr x30, [sp, #248]\n\t"  27 "add sp, sp, #256\n\t"  28 : // No output operands  29 : "r" (STACK\_START\_PTR\_ADDR), "r" (ENTRY\_PTR\_ADDR)  30 :  31 );  32 xil\_printf("\r\nELF DONE\r\n");  33 clear\_DRAM(); |

### 3.1.2.Board Initialization

현재의 경우 PS는 APU를 이용해 구동중이며 UART를 이용해 PC에 화면에 출력하거나, TCP를 이용해 PC로부터 명령, 혹은 ELF파일을 받아 수행하는 작업을 한다. 이전에 TCP를 통해 명령을 전송하면 처리하는 기능이 있었으나 PL side의 설계가 변경됨에 따라 현재는 이용할 수 없는 기능이다. TCP통신의 경우 Xilinx에서 제공하는 LWIP라이브러리를 이용해 수행하며 initialization()함수를 이용해 초기화한다. initialization() 내부에서 아래의 코드를 통해 IP를 지정하게 된다.

|  |
| --- |
| 01 /\* initialize IP addresses to be used \*/  02 IP4\_ADDR(&ipaddr, 192, 168, 1, 10);  03 IP4\_ADDR(&netmask, 255, 255, 255, 0);  04 IP4\_ADDR(&gw, 192, 168, 1, 1);  05 xil\_printf("set manual IP\n\r");  06  07 print\_app\_header();  08  09 lwip\_init();  10 xil\_printf("lwip initialize \n\r"); |

이후에는 아래의 코드를 통해 RF에 공급되는 클락을 지정한 후 set\_current\_binary\_mode함수를 통해 PC로부터 받아들이는 정보가 ELF의 binary 파일이 아닌 command임을 지정한다.

|  |
| --- |
| 01 /\* start the application (web server, rxtest, txtest, etc..) \*/  02 start\_application();  03 set\_clock(2000000);  04 /\* set binary mode false\*/  05 set\_current\_binary\_mode(0); |

### 3.1.3.Command Set

TCP를 통해 FPGA로 전달하는 명령어는 simple\_lexer.c의 simple\_lexer함수에서 처리하게 된다. 명령어의 구조는 아래와 같이 크게 세가지의 형태를 띠게 된다. 세개의 형태 모두 공통적으로 ‘#’문자를 통해 토큰을 분리하게 된다. 이 때 ‘!EOL’ 토큰을 통해 커맨드의 마지막임을 나타낸다.

|  |
| --- |
| 1. Timestamp output format  #{Module name}#{Function name}#{Timestamp}#{Param}#!EOL    2. CPU instruction format  #CPU#{Function name}#{Param}#!EOL  3. ELF related process format  #BIN#{Function name}#{Param = Total page num}#!EOL |

첫번째의 경우 첫번째 토큰은 모듈의 이름을 나타낸다. 예로 ‘DAC00’, ‘TimeController’와 같이 PL side에 구현된 모듈의 이름을 나타낸다. 두번째 토큰의 경우 해당 모듈에서 실행할 작업, 그리고 세번째 토큰의 경우 해당 작업을 수행할 timestamp, 그리고 그 다음은 해당 작업에 필요한 부가적인 parameter를 포함하고 있다. 예를 들어 ‘#DAC00#write\_fifo#10#0x11011#!EOL’의 경우 DAC00 모듈에 10 cycle에 0x11011의 output을 출력하도록 FIFO에 입력하는 것이다. 하지만 해당 명령어의 경우 PL의 구조가 지속적으로 변화함에 따라 사용하지 않으며 대부분의 명령어는 컴파일한 ELF binary code에 포함되어 있다.

두번째는 PL side에서 처리하는 작업이 아닌 주로 CPU에서 처리하는 작업을 전송할 때 사용하는 command이다. 예를 들어 “#CPU#set\_clock#4000000#!EOL”의 경우 RF clock을 4GHz로 지정한다. 해당 명령어에 해당하는 코드들은 아래에 정리되어 있다.

|  |
| --- |
| #CPU#set\_clock#{clock frequency}#!EOL  클락 주파수를 지정하는 command이다. {clock frequency}에 원하는 주파수를 입력하면 된다. 단, clock\_config.c의 XClockingLmx ClockingLmx[MAX\_FREQ] 배열에 선언되어 있는 주파수만이 가능함에 유의한다.  #CPU#read\_sampling\_freq#!EOL  설정된 RF sampling frequency를 읽은 뒤 UART통신을 통해 출력한다. |

세번째는 ELF파일을 저장, 실행하는 작업을 하는 명령어이다. 각각의 명령어는 아래에 정리되어 있다.

|  |
| --- |
| #BIN#save\_binary#{entry\_point}#{stack\_start}#{stack\_end}#{heap\_start}  #{heap\_end} #{length of data\_packets}#!EOL  ELF파일의 binary code를 DRAM에 저장하는 명령어이다. 해당 명령어를 입력한 이후 binary mode가 1로 바뀐 후 #0x16#0x72…#!EOL 형태의 데이터를 {length of data\_packets}만큼 전송한다. 모든 데이터가 전송된 이후에는 binary mode가 다시 0으로 초기화 된다.  #BIN#run\_binary#!EOL  DRAM에 저장되어 있는 ELF파일로 jump하여 ELF파일을 실행한다. |

# 4. Experiment Software

## 4.1.Compiler

Sequencer같은 경우 어셈블리어를 통해 CPU에서 구동할 명령어을 UART 통신을 통해 직접 입력하는 방식을 사용하였다. 하지만 RFSoC의 경우 C 언어로 작성된 코드가 CPU에서 run하면서 realtime으로 FIFO로 instruction을 output하는 방식이다. 작성된 C Code는 GNU 컴파일러를 이용해 컴파일하게 된다.

컴파일러는 Vivado\_prj\_manager 프로젝트[[9]](#footnote-9)에서 Compiler\PythonCode 디렉토리의 C\_compiler.py에 존재한다. 해당 파이썬 파일이 자동으로 컴파일 스크립트를 만들어 실행하여 C 파일들을 모두 컴파일한다. GNU 컴파일러와 기타 필요한 설치파일들은 Compiler\GNU\_Installer에 존재하며 해당 파일들을 모두 설치해야 C\_compiler.py이 정상적으로 작동한다.

파일 내 주요 함수들은 아래와 같다.

|  |
| --- |
| C\_Compiler.py  compileCode  C 파일들을 찾은 뒤 Xilinx에서 제공하는 라이브러리와 RFSoC Driver와 함께 ELF파일로 컴파일한다. 컴파일 과정에서 자동으로 라이브러리를 함께 컴파일하게 되므로 실험 코드를 작성할 때 수동으로 Xilinx 라이브러리 파일들이나 RFSoC Driver들을 포함할 필요가 없다.  createTCPpacket  컴파일된 ELF파일을 RFSoC에 전송할 수 있는 데이터 리스트로 변경한다. |

## 4.2.PL Modules

PL 모듈들을 드라이버 코드들이 Vivado\_prj\_manager 디렉토리를 기준으로 Compiler\C\_Code\RFSoC\_Driver 에 작성되어 있다. Compiler\C\_Code 에 컴파일할 디렉토리를 만들고 동일한 이름으로 C파일을 작성한 후 C\_compiler.py으로 컴파일하여 RFSoC로 전송하는 방식이다. 예를들어 TEST1.cpp이라는 파일을 TEST1디렉토리를 만든 후 해당 디렉토리내에서 작성하면 된다. 모든 모듈들은 객체 기반으로 작동한다. 따라서 DAC::set\_addr과 같은 함수명은 DAC객체에 대한 함수를 나타낸다.

### 4.2.1.DAC

|  |
| --- |
| DAC.cpp  void DAC::set\_addr(uint64\_t addr)  DAC의 AXI 주소를 지정한다.  void DAC::set\_freq(uint64\_t freq)  DAC의 output주파수를 지정한다. 48bit의 resolution을 가진다.  void DAC::set\_amp(long double amp)  DAC의 output 진폭을 지정한다. 최대 amplitude에 대한 비율을 input으로 받으며 14bit의 resolution을 가진다.  void DAC::set\_config(long double amp, uint64\_t freq,  long double phase, uint64\_t shift)  DAC의 output 진폭, 주파수, 그리고 위상을 한번에 변경한다. 진폭, 위상, 주파수 각각 14, 14, 32 bit의 resolution을 가진다. 단,. shift의 두배만큼 frequency를 shift한 bit에 해당하는 주파수를 변경하며 그 외의 bit들은 모두 무시한다. 예를들어 shift = 3인 경우 48bit의 주파수 중 37~6 bit에 해당하는 값만을 변경하고 그 외의 bit들은 유지한다.  void DAC::flush\_fifo()  DAC 모듈의 fifo를 초기화한다. |

### 4.2.2.TTL

|  |
| --- |
| TTL\_out.cpp  void TTL\_out::set (uint64\_t pulse)  TTL신호의 output을 지정한다. pulse = 1인 경우 High, pulse = 0인 경우 Low를 출력한다. 단. 8ns의 resolution을 가지므로 8ns이하의 1ns의 차이는 모두 무시된다. 예를 들어 35ns에 output을 지정하는 경우 35 = 0x13이므로 0x03은 무시한 후 32ns에 output이 출력된다.  void TTL\_out::set\_ch(const char \* pulse\_ch)  TTL신호의 output을 지정한다. int형 대신 string을 입력받는다. 대소문자 관계없이 “ON”인 경우 High, “OFF”인 경우 Low를 출력한다.    void TTL\_out::flush\_fifo()  TTL 모듈의 fifo를 초기화한다. |

### 4.2.3.TTLx8

|  |
| --- |
| TTLx8.cpp  void TTLx8\_out::set (uint64\_t pulse)  TTL신호의 output을 지정한다. 1ns의 resolution을 가진다.    void TTLx8\_out::set\_ch(uint64\_t freq)  TTL신호의 output을 지정한다. int형 대신 string을 입력받는다. 대소문자 관계없이 “ON”인 경우 High, “OFF”인 경우 Low를 출력한다.  void TTLx8\_out::flush\_fifo(long double amp)  TTLx8 모듈의 fifo를 초기화한다 |

### 4.2.4.TimeController

|  |
| --- |
| TimeController.cpp  void TimeController::reset ()  TimeController의 timestamp를 0으로 초기화한다..    void TimeController::auto\_start()  TimeController의 timestamp를 증가시키기 시작하고 모든 모듈에 FIFO에 저장된 instruction을 output하도록 한다.  void TimeController:: auto\_stop()  TimeController의 timestamp 증가를 중지시키고 모든 모듈들에서 FIFO에 저장된 instruction을 output하지 않도록 한다. |

# 5. Project Creation

## 5.1.Vivado Creation

## 5.2.Vitis Creation

1. ARTIQ을 개발한 M-labs의 경우에도 Zynq7000 series의 hardcore 방식의 board가 존재하며 2023년을 기점으로 RFSoC기반의 보드가 개발중에 있다. [↑](#footnote-ref-1)
2. Crockett, L. H., Northcote, D., &amp; Stewart, R. W. (2023). Software defined radio with Zync UltraScale+ rfsoc. Strathclyde Academic Media. [↑](#footnote-ref-2)
3. An Adaptable Direct RF-Sampling Solution. AMD Adaptive Computing Documentation Portal. (n.d.). <https://docs.xilinx.com/v/u/en-US/wp489-rfsampling-solutions> [↑](#footnote-ref-3)
4. TICSPRO-SW. TICSPRO-SW Application software &amp; framework | TI.com. (n.d.). [https://www.ti.com/tool/TICSPRO-SW#downloads](https://www.ti.com/tool/TICSPRO-SW" \l "downloads) [↑](#footnote-ref-4)
5. Xilinx 설치 위치 기준으로 Xilinx\Vitis\2020.2\data\embeddedsw\XilinxProcessorIPLib\drivers\rfdc\_v8\_0\examples\xrfdc\_clk.c 을 이용한다.. [↑](#footnote-ref-5)
6. Introduction to Amba Axi4 - arm developer. (n.d.). <https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/102202_0100_01_Introduction_to_AMBA_AXI.pdf?revision=369ad681-f926-47b0-81be-42813d39e132> [↑](#footnote-ref-6)
7. Erdmann, C., Cullen, E., Brouard, D., Pelliconi, R., Verbruggen, B., Mcgrath, J., Collins, D., De La Torre, M., Gay, P., Lynch, P., Lim, P., Collins, A., & Farley, B. (2017, February). 16.3 A 330mW 14b 6.8GS/s dual-mode RF DAC in 16nm FinFET achieving −70.8dBc ACPR in a 20MHz channel at 5.2GHz. 2017 IEEE International Solid-State Circuits Conference (ISSCC). https://doi.org/10.1109/isscc.2017.7870370 [↑](#footnote-ref-7)
8. Razavi, B. (2018). The Current-Steering DAC [A Circuit for All Seasons]. IEEE Solid-State Circuits Magazine, 10(1), 11–15. https://doi.org/10.1109/mssc.2017.2771102 [↑](#footnote-ref-8)
9. <https://github.com/snu-quiqcl/Vivado_prj_manager> [↑](#footnote-ref-9)