**RFSoC:LOLENC**

QuIQCL RFSoC LOw LatENCy Device Development Note

**- Written by: jhpark snu -**

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



# Contents

[Contents 2](#_Toc149405291)

[1. Introduction to RFSoC 3](#_Toc149405292)

[1.1.RFSoC Chip 3](#_Toc149405293)

[1.2.ZCU111 3](#_Toc149405294)

[1.3.AXI Bus 5](#_Toc149405295)

[2. PL Side Architecture 6](#_Toc149405296)

[2.1.RTO & GPO 6](#_Toc149405297)

[2.2.DAC 6](#_Toc149405298)

[2.3.TTL 6](#_Toc149405299)

[2.4.TTLx8 6](#_Toc149405300)

[2.5.TimeController 6](#_Toc149405301)

[3. PS Side Firmware 6](#_Toc149405302)

[3.1.Firmware 6](#_Toc149405303)

[3.1.1.Running ELF 6](#_Toc149405304)

[3.1.2.Command Set 8](#_Toc149405305)

[4. Experiment Side Software 10](#_Toc149405306)

[4.1.Compiler 10](#_Toc149405307)

[4.2.Devices 10](#_Toc149405308)

[4.2.1.DAC 10](#_Toc149405309)

[4.2.2.TTL 10](#_Toc149405310)

[4.2.3.TTLx8 10](#_Toc149405311)

[4.2.4.TimeController 10](#_Toc149405312)

[5. Project Creation 10](#_Toc149405313)

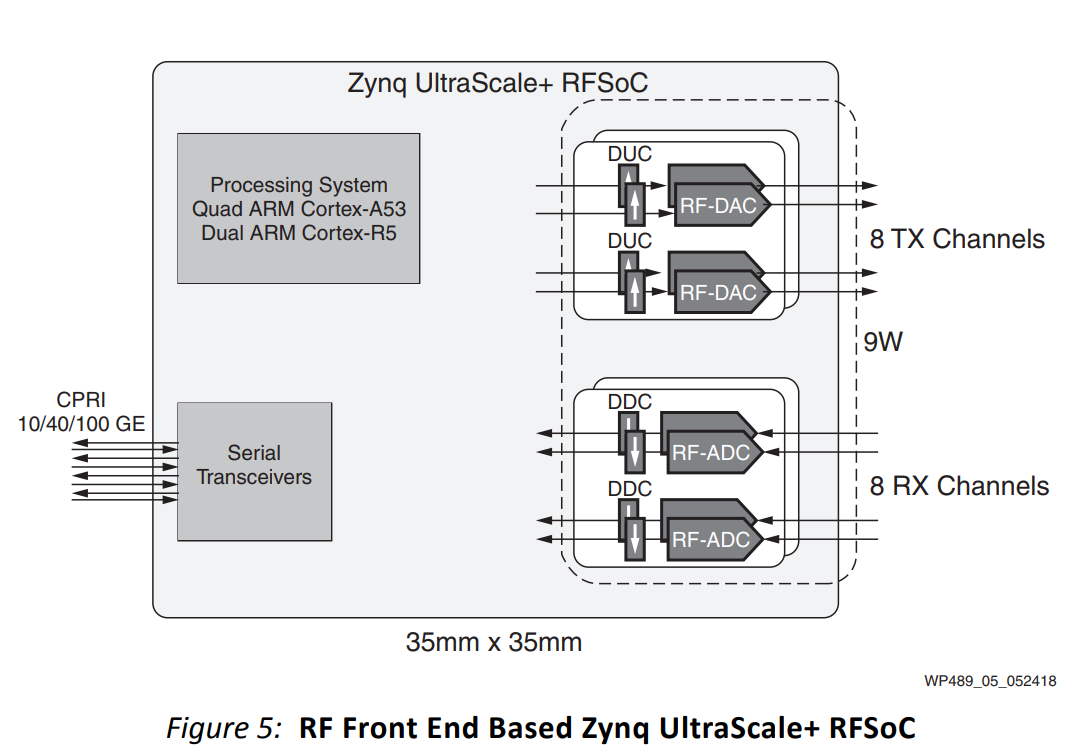
[5.1.Vivado Creation 10](#_Toc149405314)

[5.2.Vitis Creation 10](#_Toc149405315)

# 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 Gen3에 갈수록 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.



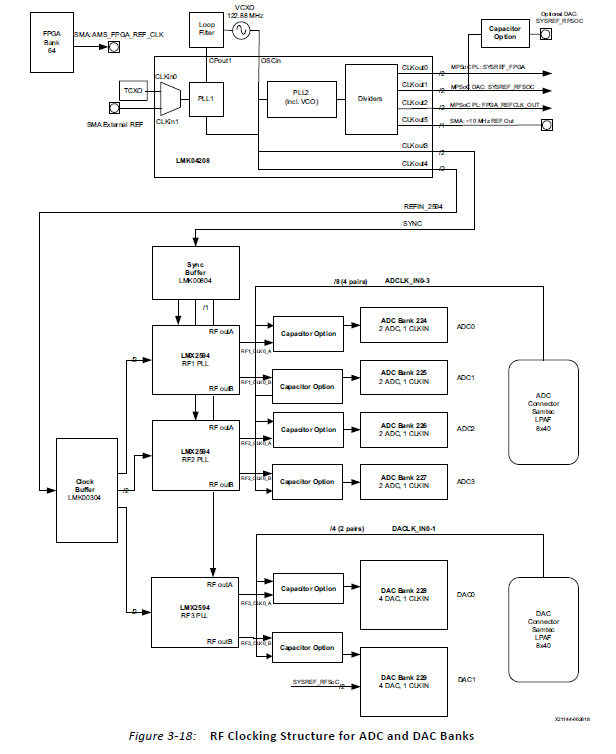
RFSoC의 경우 AMD의 CPU가 내장되어 있는 hardcore방식에 해당한다. ARTIQ의 경우 OpenRISC시리즈를 기반으로 한 mor1kx를 사용하는 것과 대비된다.[[1]](#footnote-1) CPU의 경우 Application Processing Unit(APU), 그리고 Realtime Processing Unit(RPU)로 구성된다. XCZU28DR의 경우 4개의 APU, 그리고 2개의 RPU로 구성된다. APU의 경우 통상적인 CPU로 생각할 수 있으며 ARMv7와는 다르게 64bit architecture이며 기존의 ARMv7과는 다른 exception level을 사용한다. RFSoC에 관한 더 자세한 내용은 해당 문서를 참고 바란다.[[2]](#footnote-2)[[3]](#footnote-3)

## 1.2.ZCU111

RF신호는 RFMC, TTL 신호의 경우 FMC(VITA57.4)으로부터 입출력되게 된다. 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 제한된다.

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 시리얼 통신을 통해 변경해야 한다. 또한 RF 클락의 주파수를 변경하는 경우에도 마찬가지의 작업을 해야 한다.

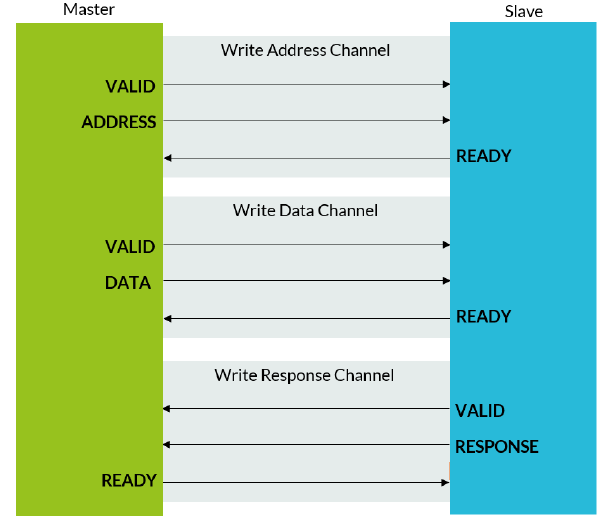
RF클락의 경우 아래와 같은 회로에서 출력이 된다. LMK04028와 LMX2594은 PLL, LMK00304는 클락 버퍼에 해당한다. Xilinx에서 제공하는 펌웨어에서는 LMK04028을 122.88MHz의 oscillator를 입력받아 다시 122.88MHz의 클락을 출력하고 LMX2594에서 곱해진 주파수의 클락을 출력한다. TICS Pro 프로그램[[4]](#footnote-4)을 이용해 GUI상에서 주어진 주파수의 시리얼 통신 데이터를 만들 수 있으며 주어진 시리얼 통신 데이터는 Xilinx에서 주어지는 example code를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



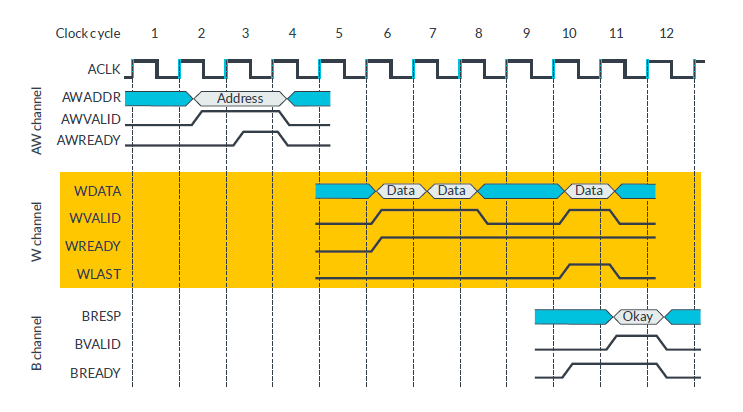
보드와 PC사이에서 이더넷 통신 및 UART통신이 모두 가능하다. 현재 FPGA를 프로그래밍하는 경우와 FPGA로부터 출력 결과를 PC에서 받는 경우에는 시리얼 통신을 수행하며 명령어를 입력은 이더넷을 이용한다.

## 1.3.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)로 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)



# 2. PL Side Architecture

## 2.1.RTO & GPO

## 2.2.DAC

## 2.3.TTL

## 2.4.TTLx8

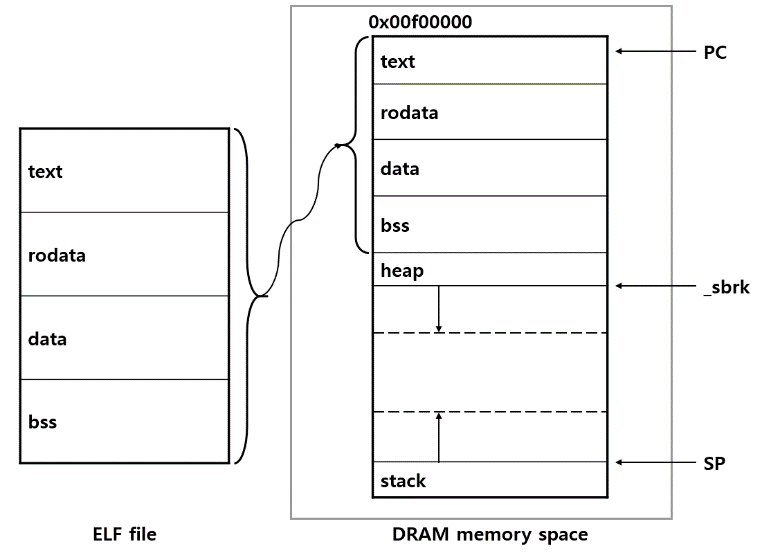
## 2.5.TimeController

# 3. PS Side Firmware

## 3.1.Firmware

### 3.1.1.Running ELF

실험을 위해 작성된 코드를 컴파일하기 위해 gnu-aarch64-none-elf-g++ 컴파일러를 이용하여 ELF(Executable Loadable File)로 컴파일 한 후 FPGA(RFSoC)에 데이터를 전송하여 DRAM의 **0x00f00000**부터 저장한 후 CPU의 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.



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.S**파일을 포함하고 linker 파일 **linker.ld**에서 entry point를 **\_start\_custom\_**으로 지정하였다. **startup.S**는 BSS 영역에 해당하는 변수들을 모두 0으로 초기화 한후 main함수로 jump한다.

ELF를 실행한 후 원래의 함수로 return하기 위해 기존의 레지스터에 존재하던 모든 값들을 stack에 저장한다. 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.c**에서 수행하며 아래에 주어져 있다.

|  |
| --- |
| **Xil\_DCacheFlush();**  **Xil\_ICacheInvalidate();**  **\_\_asm\_\_ \_\_volatile\_\_ (**  **"sub sp, sp, #256\n\t"**  **"str x0, [sp, #8]\n\t"**  **…**  **"str x30, [sp, #248]\n\t"**  **/\*save SP\*/**  **"mov x0, sp\n\t"**  **"mov x2, #0x700000\n\t"**  **"str x0, [x2]\n\t"**  **/\*save END\*/**  **"ldr x1, [%0]\n\t" // Load sp value to x1**  **"mov sp, x1\n\t" // Move sp value to stack pointer (sp)**  **"ldr x1, [%1]\n\t" // Load main address to x1**  **/\*####JUMP TO MAIN####\*/**  **"blr x1\n\t" // Branch to the address in x1 (MAIN function)**  **/\*return SP\*/**  **"mov x2, #0x700000\n\t"**  **"ldr x0, [x2]\n\t"**  **"mov sp, x0\n\t"**  **/\*return END\*/**  **"ldr x0, [sp, #8]\n\t"**  **"ldr x1, [sp, #16]\n\t"**  **…**  **"ldr x30, [sp, #248]\n\t"**  **"add sp, sp, #256\n\t"**  **: // No output operands**  **: "r" (STACK\_START\_PTR\_ADDR), "r" (ENTRY\_PTR\_ADDR)**  **:**  **);**  **xil\_printf("\r\nELF DONE\r\n");**  **clear\_DRAM();** |

### 3.1.2.Command Set

현재의 경우 PS는 APU를 이용해 구동중이며 UART를 이용해 PC에 결과를 송출하거나, 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 LWIP을 이용해 수행하며 main.c내부에서 아래의 코드를 통해 IP를 지정하게 된다.

|  |
| --- |
| **/\* initialize IP addresses to be used \*/**  **IP4\_ADDR(&ipaddr, 192, 168, 1, 10);**  **IP4\_ADDR(&netmask, 255, 255, 255, 0);**  **IP4\_ADDR(&gw, 192, 168, 1, 1);**  **xil\_printf("set manual IP\n\r");**  **print\_app\_header();**  **lwip\_init();**  **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 comand임을 지정한다.

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

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.c**의 **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 Side Software

## 4.1.Compiler

## 4.2.Devices

### 4.2.1.DAC

### 4.2.2.TTL

### 4.2.3.TTLx8

### 4.2.4.TimeController

# 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 보드를 개발하는 중이다. [↑](#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> [↑](#footnote-ref-4)
5. Xilinx 설치 위치 기점으로 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)