

목차

1. RV32I Multicycle 구조 요약

1.1. Multicycle CPU 설계 목적

1.1.1. Multicycle CPU 단계

1.2. Memory Stage에서 APB BUS 사용 설명

1.2.1. Memory Access 접근 명령어

2. AMBA APB BUS Protocol

2.1. APB Protocol Introduction

2.2. Signal Descriptions

2.2.1. AMBA APB signals

2.2.2. Address bus

2.2.3. Data buses

2.3. Transfer

2.3.1. Write transfers

2.3.2. Read transfers

2.4. Operating States

3. Peripheral

3.1. FND Controller

3.1.1. Introduction

3.1.2. FND main features

3.1.3. FND register map

3.1.4. FND Interface Description

3.1.5. Peripheral Operation

3.1.6. C Language Access Example

3.2. General Purpose I/Os (GPIO)

3.2.1. Introduction

3.2.2. GPIO main features

3.2.3. GPIO registers

3.3. Ultrasound Peripheral

- 3.3.1. Introduction**
- 3.3.2. Main Features**
- 3.3.3. Functional Description**
- 3.3.4. 설계 상세 내용**
- 3.3.5. Ultrasound Peripheral 검증**

3.4. Universal Asynchronous Receiver Transmitter (UART)

- 3.4.1. UART introduction**
- 3.4.2. UART main features**
- 3.4.3. UART functional description**
- 3.4.4. UART baud rate generation**
- 3.4.5. UART register**
- 3.4.6. Asynchronous Data Reception**

4. Firmware Implementation

4.1. Introduction

4.2. 시스템 구성

4.3. 동작 시나리오

- 4.3.1. 수동 측정 모드 (Manual Mode)**
- 4.3.2. 자동 측정 모드 (FSM 기반)**
- 4.3.3. FND 출력 선택**

4.4. 주요 코드 및 제어 로직

- 4.4.1. FSM 기반 UART 명령 처리**
- 4.4.2. 타이머 모듈을 활용한 주기적 센서 측정**

1. RV32I Multicycle 구조 요약

1.1. Multicycle CPU 설계 목적

Singlecycle CPU (RISC-V, RV32I)의 경우 모든 명령어 종류들을 각각 실행하는데 1 CLK이 소요된다. Core가 ROM에서 명령어를 읽으면 Control Unit이 Combinational Logic으로 구성되어 있어 DataPath에 Control Signal을 내보내고 그에 맞추어 DataPath가 동작하기 때문이다.

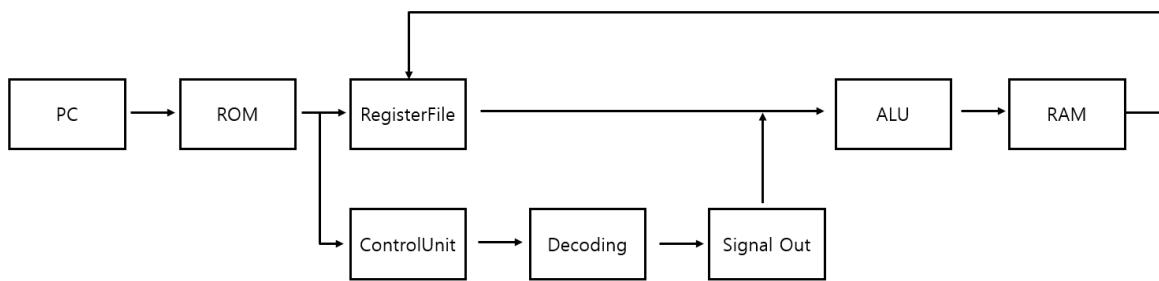


그림 1

하지만 이러면 가장 오래 걸리는 명령어(L-type)의 실행 시간에 맞추어 CLK 주기를 설정해야 하므로 길 수 밖에 없다. 따라서 CLK의 주기를 낮추어 단계를 나누어 구성 할 것이다.

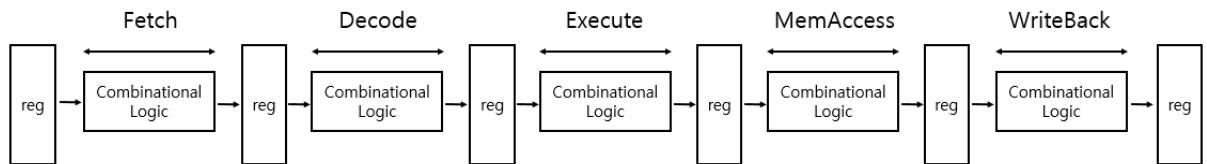


그림 2

그림 1과 같이 5단계, Fetch, Decode, Execute, MemAccess(Memory Access), WriteBack으로 이루어졌다.

각 단계 사이에는 값을 유지해주기 위한 register를 구성했다.

1.1.1. Multicycle CPU 단계

Fetch : 명령어를 인출하는 단계로 CPU는 ROM에서 명령어를 가져온다.

Decode : 명령어를 분석하는 단계로 Control Unit에서 이루어진다.

이 때 PC에는 singlecycle과 다르게 enable 신호가 필요하다. 왜냐하면 모든 명령어가 끝날 때까지 다음 명령어가 출력이 되면 안되기 때문이다.

Execute : 연산에 대한 실행을 하는 단계로 ALU연산을 실행한다.

MemAccess : RAM에 신호를 전달하는 단계로, address신호, data 신호등을 전달한다.

WriteBack : RAM의 데이터를 읽어오는 단계다.

1.2 Memory Stage에서 APB BUS 사용 설명

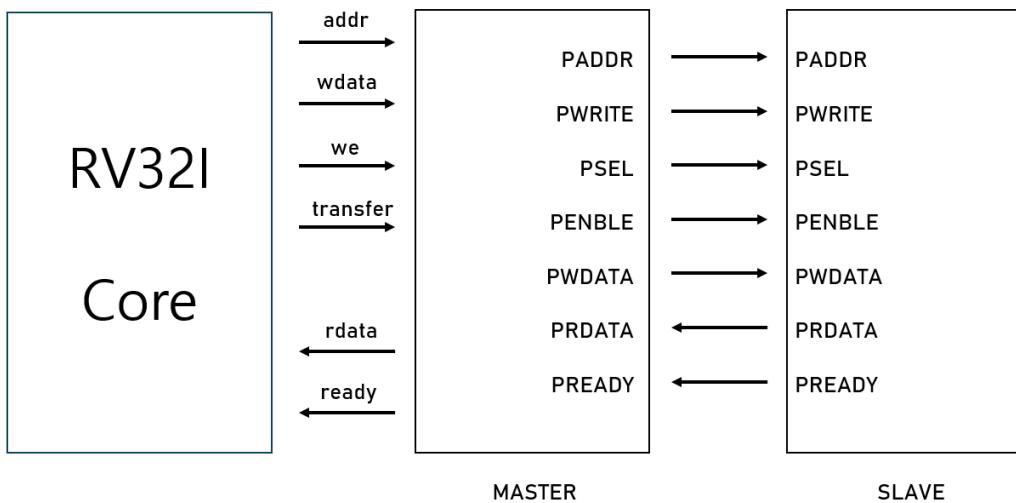


그림 3

CPU가 MemoryAccess 단계에 접근하면 RV32I Core에서 transfer신호를 APB Master에게 전달한다. 이 때 Core에서 Write을 할 거면 address, wdata, we(write enable)을, Read를 할 거면 address, rdata, we(=0)을 Master에게 주며 Write, Read를 한다. Master는 transfer 신호를 받으면 IDLE → SETUP → ACCESS 순서로 APB Master의 state가 변화한다.

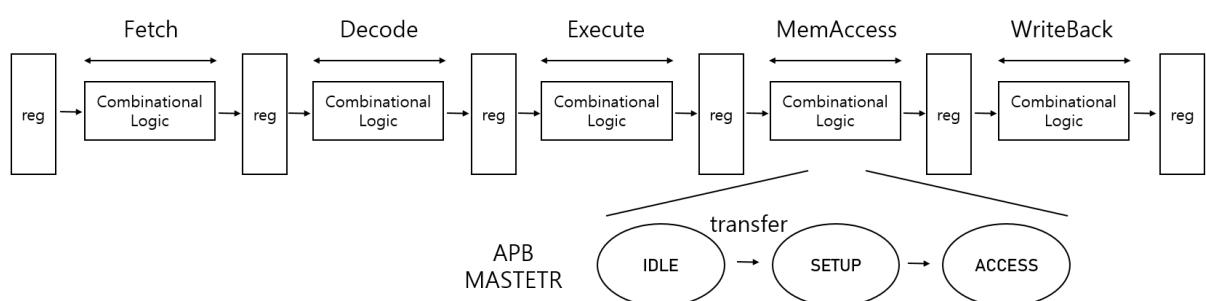


그림 4

IDLE : transfer 신호가 들어오면 Core로부터 address, wdata, write을 받아오며 SETUP state로 가게 된다.

SETUP : 받아온 address를 바탕으로 Slave를 선택하고 그 Slave의 register에 접근할 수 있는 준비를 한다. 그리고 we가 1이면 wdata를 준비하여 write을 하게되고, 0이면 read를 진행한다. 그리고 ACCESS state로 가게 된다.

ACCESS : PENABLE이 1이 되면서 SETUP에 준비했던 data를 기반으로 통신을 하게 된다. 그리고 slave로부터 통신이 끝났다는 PREADY를 받게 되면 Master도 CPU에게 통신이 끝났다는 ready를 전송하고 IDLE state로 가게 된다.

1.2.1. Memory Access 접근 명령어

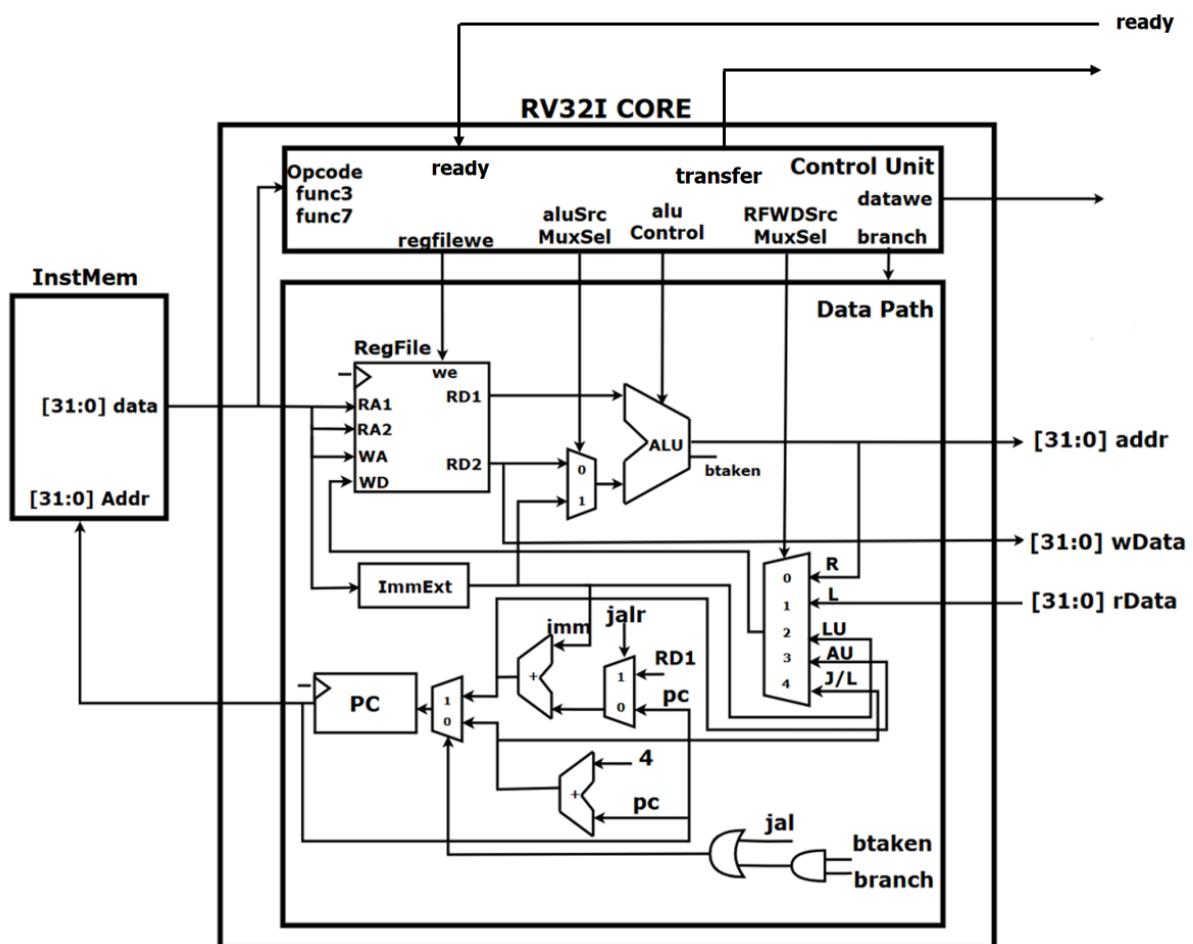


그림 5 ※RV32I Core는 Single Cycle로 대체

1.2.1.1 Load Instructions (L - type)

메모리에서 값을 읽어와 레지스터에 저장하는 명령어이다.

Ex) LW (Load Word)

Inst	31	...	25	24	...	20	19	...	15	14	13	12	11	...	7	6	...	0	TYPE	MNEMONIC
	Imm(12)					Source			Funct(3)			Destination			Opcode					
	Imm[11:0]					rs1			0 0 0			rd			0000011			L	Load Word	

표 1

Descript : $rd = M[rs1 + imm][0:31]$

a) MemAccess

그림 5에서 볼 수 있듯이, InstMem은 명령어를 Control Unit과 DataPath로 전달한다.

DataPath에서 rs1($=Inst[19:15]$)과 imm[11:0]을 ALU 연산을 통해서 접근할 메모리의 주소([31:0] addr)를 구하고 APB MASTER에게 전달한다.(그림 3 참고).

Control Unit에서는 we(datawe =0)와 transfer 신호를 APB Master에게 전달한다.(그림 3 참고).

APB MASTER는 전달받은 주소를 어떤 Slave에 접근하고, 그 Slave의 어느 register에 접근할지를 결정한다. Master가 Slave에서 Read를 수행하면, 읽은 값인 rdata를 Core에 보내며 동시에 ready 신호를 전달한다.

Core는 ready를 수신하면 WriteBack 단계로 간다.

b) WriteBack

Control Unit에서 transfer를 0으로 설정하고, regfilewe를 1로 설정하여 전달받은 rdata를 registerfile 저장한다. 그 후, Fetch 단계로 돌아가 다음 명령어를 실행할 준비를 한다.

1.2.1.2 Store Instructions (S - type)

레지스터에 있는 값을 메모리에 저장하는 명령어이다.

Ex) SW (Store Word)

Inst	31	...	25	24	...	20	19	...	15	14	13	12	11	...	7	6	...	0	TYPE	MNEMONIC
	Imm7		Source2			Source1			Funct(3)			Imm5			Opcode					
	Imm[11:5]		Rs2			Rs1			0 0 0			Imm[4:0]			0100011			S	Store Word	

표 2

Descript : $M[rs1 + imm][0:31] = rs2[0:31]$

a) MemAccess

그림 5에서처럼 InstMem은 명령어를 Control Unit과 DataPath로 전달한다.

DataPath에서는 rs1(=Inst[19:15])과 imm[11:0](={Inst[31:25], Inst[11:7]})을 사용해 ALU 연산을 수행하고, 접근할 메모리의 주소([31:0] addr)를 구한 뒤 APB MASTER에게 전달한다.(그림 3 참고).

Control Unit은 wdata, we(datawe =1)그리고 transfer 신호를 APB Master에게 전달한다.(그림 3 참고).

APB MASTER는 전달받은 주소를 어떤 Slave에 접근하고, 그 Slave의 어느 register에 접근할지를 결정한다. Master가 Slave에 Write을 수행하고 Slave로부터 PREADY를 수신하면 ready 신호를 Core로 전달한다.

Core는 ready를 받으면 Fetch 단계로 이동한다.

2. AMBA APB BUS Protocol

2.1 APB Protocol Introduction

APB는 peripheral을 제어하는데 특화된 저전력·저비용 버스이다. Peripheral의 programmable register를 제어(read/write)하는 용도로 사용되며 APB Peripheral은 APB bridge를 통해 main memory system에 연결된다.

APB 전송은 APB bridge에 의해 시작되며, 이 bridge를 Master라 한다. Peripheral Interface는 Master에 응답하며, Slave라 통칭한다.

2.2 Signal Descriptions

2.2.1 AMBA APB signals

일부 신호는 width가 고정되어 있고, 일부 신호는 width를 조정할 수 있다.

Signal	Source	Width	Description
PCLK	Clock	1	Clock PCLK은 clock signal이며, 모든 APB signal은 PCLK의 상승에 기준으로 동작한다.
PRESET	System bus reset	1	Reset RESET은 시스템 버스 리셋 신호에 직접 연결된다.
PADDR	Master	32bit	Address PADDR는 APB 주변장치의 레지스터에 접근하기 위한 주소 신호이다. Master가 접근하고자 하는 Register 주소를 지정한다.
PSEL	Master	1	Select Peripheral을 선택하기 위해 생성한다. 해당 Peripheral이 선택되었고 데이터 전송이 요구됨을 의미한다.
PENABLE	Master	1	Enable
PWRITE	Master	1	Direction(전송 방향 결정) HIGH이면 write 동작, LOW이면 read 동작을 나타낸다.
PWDATA	Master	32bit	Write data PWRITE가 HIGH일 때 구동되며 DATA 폭은 8, 16 또는 32 bit로 설정 가능하다.
PREADY	Slave	1	Ready Slave가 APB Transfer를 연장하는 데 사용된다. (PREADY LOW로 인해 시 이후 High로 조정될 때까지 Transfer가 연장됨)
PRDATA	Slave	32bit	Read data

			Master는 해당 버스를 통해 Register data를 수신하며 PWRITE 이 LOW일 때 동작한다.
--	--	--	--

2.2.2 Address bus

APB Interface는 read/write 연산 수행을 위해 단 하나의 주소 버스(PADDR)만을 사용하며, PADDR은 byte 단위의 주소를 나타낸다. Unaligned address도 사용이 가능하지만, 결과는 예측할 수 없다. (unpredictable)

2.2.3 Data buses

APB protocol은 8, 16, 또는 32 bit 폭의 독립된 2개의 버스로 구성되어 있으며, 하나는 read data 용이고 나머지 하나는 write data 용이다. Read bus와 write bus는 동일한 width를 가져야하며, 읽기와 쓰기 데이터 전송은 각 버스에 독립적인 handshake 신호가 존재하지 않기에 동시에 수행될 수 없다.

2.3 Transfer

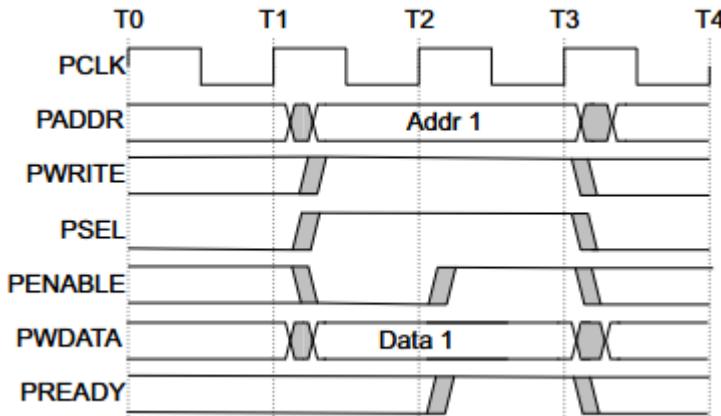
2.3.1 Write transfers

이 섹션에서는 아래 두 유형의 쓰기 전송(write transfer)에 대해 설명한다.

- 대기 상태(wait state)가 없는 전송
- 대기 상태가 있는 전송

이 섹션의 모든 신호는 PCLK의 상승 에지에서 샘플링된다.

2.3.1.1 With no wait states

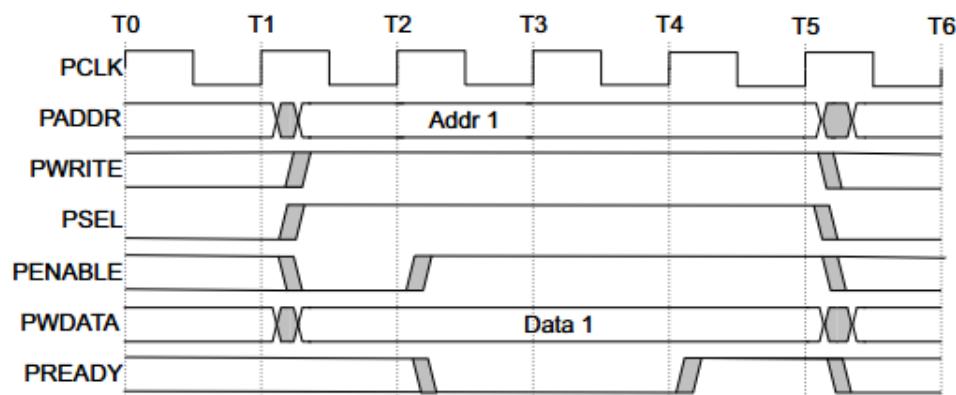


Write Transfer의 Setup stage는 T1 시점에 발생한다. 이 때 PSEL이 활성화 되며, 이 시점의 PADDR, PWRITE, PWDATA는 모두 유효한 값으로 setting되어야 한다.

T2 시점에 PENABLE이 활성화되며 Write Transfer의 ACCESS 단계가 시작된다. 이후 T3 시점의 PCLK의 상승 에지에서 Slave가 해당 Write data가 수신될 것임을 알린다. 이 시점부터 전송이 완료될 때까지 PADDR, PWDATA 및 기타 제어 신호들은 Stable 상태 여야 하며, 전송이 완료되면 PENABLE이 deasserted 되고 PSEL 또한 비활성화된다.

2.3.1.2 With wait states

그림 xx는 Slave가 PREADY LOW로 유지하여 transfer를 연장할 수 있음을 보여준다.



ACCESS 단계에서, PENABLE이 HIGH인 동안 Slave가 LOW를 유지하여 transfer를 연장할 수 있다.

이 때 PADDR, PWRITE, PSEL, PENABLE, PWDATA는 변경되지 않고 유지되어야 한다.

(PENABLE이 LOW이면 PREADY가 어떤 값이든 상관없다.)

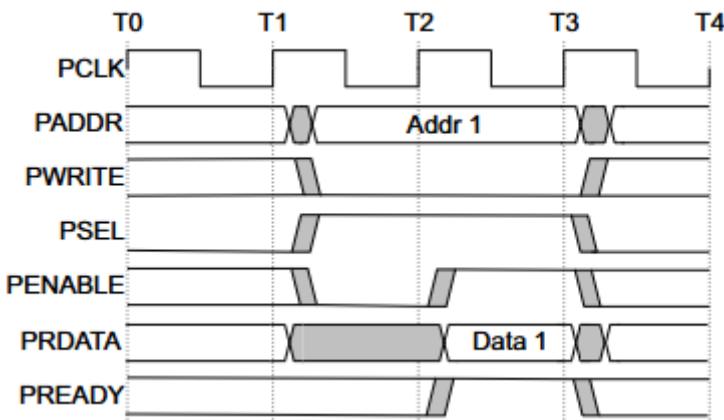
2.3.2 Read transfers

이 섹션에서는 아래 두 유형의 읽기 전송(read transfer)에 대해 설명한다.

- 대기 상태(wait state)가 없는 전송
- 대기 상태가 있는 전송

이 섹션의 모든 신호는 PCLK의 상승 에지에서 샘플링된다.

2.3.2.1 With no wait states



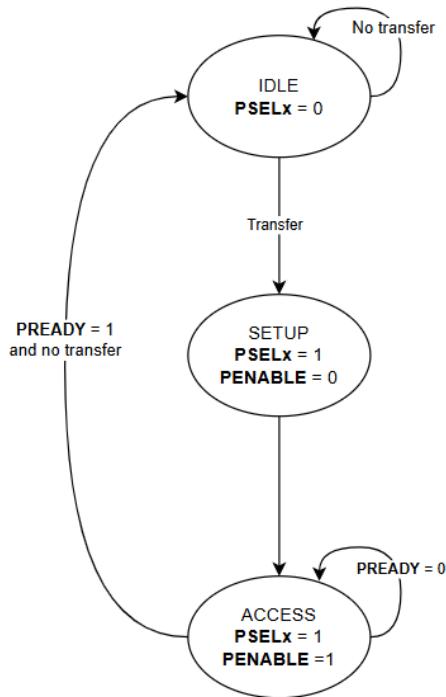
PADDR, PWRITE, PSEL, PENABLE 타이밍은 write transfer과 동일하다. Slave는 read transfer가 끝나기 전에 read data를 제공해야 한다.

2.3.2.2 With wait states



ACCESS 단계에서 PREADY가 LOW로 구동 되면 transfer이 연장된다. 이 때 PADDR, PWRITE, PSEL, PENABLE, PRDATA는 변경되지 않고 유지되어야 한다.

2.4 Operating States



State machine은 다음 state로 구동된다.

IDLE APB interface의 default state

SETUP 전송이 필요할 때 interface는 SETUP state로 진입하며 이때 특정 Peripheral을 선택하기위한 PSEL이 활성화된다. Interface는 항상 1 클럭 사이클 동안만 SETUP 상태에 머무르며, 다음 클럭 상승 에지에서 ACCESS 상태로 전이된다.

ACCESS PENABLE 신호는 ACCESS 상태에서 활성화된다. PADDR, PWRITE, PWDATA(write transaction일 경우) 는 ACCESS 상태에서 여러 클럭 사이클이 소모될 때에도 절대 변경되어서는 안된다.

ACCESS 상태의 종료 시점은 Slave가 제공하는 PREADY 신호에 의해 결정된다:

- 만약 PREADY가 LOW이면, Interface는 ACCESS 상태에 계속 머무른다.
- PREADY가 HIGH가 되면 ACCESS 상태를 종료하고, IDLE로 전이한다.

3. Peripheral

3.1 FND Controller

3.1.1 Introduction

FND 모듈은 APB 프로토콜 기반으로 설계된 7-Segment Display 제어용 주변장치이다. 해당 모듈은 초음파 거리, 온도, 습도 등 다양한 값을 시각적으로 출력하기 위해 사용된다. APB 인터페이스를 통해 마스터로부터 값을 받아 세그먼트에 표시하며, 내부적으로 BCD 변환 기능을 제공한다.

3.1.2 FND main features

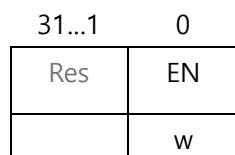
- APB 인터페이스 기반 레지스터 제어 방식
- 10진수 입력값을 자동으로 BCD로 변환하는 기능 내장
- Dot(소수점) 위치 제어 기능 지원
- 최대 8비트 세그먼트 출력 데이터 지원
- FND 출력 on/off 제어 가능

3.1.3 FND register map

offset	Register	31...8	7	6	5	4	3	2	1	0
0000	FCR									EN
0004	FDR	Reserved								FDR[13:0]
0008	FPR									DOT[3:0]
000C	NOUSE									Reserved
0010	HEX									HEX[31:0]
0014	BCD									BCD[31:0]

3.1.3.1 FND Control Register (FCR)

Address offset: 0x00



Bits 0 **EN**

해당 data는 FND 출력 기능의 활성화 여부를 제어하기 위해 software에서 작성된다.

0: 출력 비활성화 (OFF)

1: 출력 활성화 (ON)

3.1.3.2 FND Data Register (FDR)

Address offset: 0x04

31...14		13...0		
Res		FDR[13:0]		
		rw		

Bits 13:0 **FDR[13:0]**

세그먼트에 출력할 숫자 값 (0~9999)

3.1.3.3 FND Dot Register (FPR)

Address offset: 0x08

31...4 3 2 1 0				
Res	DOT[3]	DOT[2]	DOT[1]	DOT[0]
	rw	rw	rw	Rw

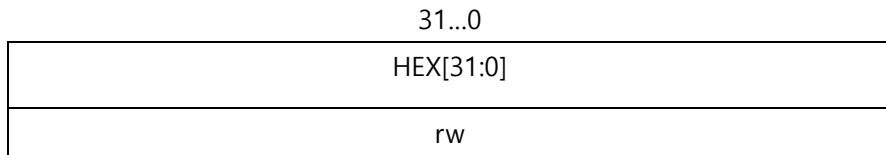
Bits 3:0 **DOT[3:0]**

FND 소수점 (dot) 출력 제어

각 비트가 1이면 해당 자릿수에 소수점 출력

3.1.3.4 Decimal Input Register (HEX)

Address offset: 0x10

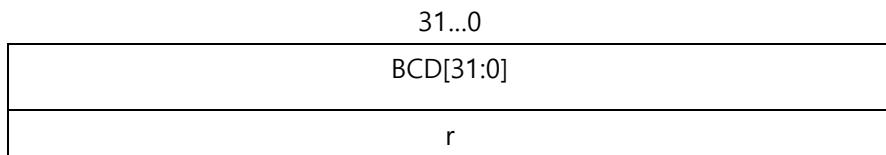


Bits 31:0 **HEX[31:0]**

정수값 입력용. 입력 시 내부 BCD 변환이 수행됨

3.1.3.5 BCD Output Register (BCD)

Address offset: 0x14



Bits 31:0 **BCD[31:0]**

HEX 레지스터에 입력된 값을 BCD로 변환한 결과를 반환

- Bit 31~24: 천의 자리 ASCII 값
- Bit 23~16: 백의 자리 ASCII 값
- Bit 15~8: 십의 자리 ASCII 값
- Bit 7~0: 일의 자리 ASCII 값

3.1.4 FND Interface Description

- FCR 레지스터를 통해 FND 출력 활성화 여부를 결정할 수 있다.
- FDR 또는 HEX 레지스터에 정수 값을 입력하면 세그먼트 출력이 수행된다.
- FPR 레지스터를 통해 dot 위치를 제어할 수 있다.
- HEX 레지스터에 값을 쓰면 내부적으로 BCD 변환을 수행하고, 결과는 BCD 레지스터에 저장된다.

3.1.5 Peripheral Operation

FND는 APB 버스를 통해 제어된다. PWRITE 신호가 HIGH인 경우에는 HEX 또는 FDR에 데이터를

write할 수 있으며, PWRITE가 LOW일 경우에는 BCD 레지스터 값을 read할 수 있다. FCR은 0x01로 설정 시 출력이 enable된다. PENABLE과 함께 APB 전송이 이루어진다.

3.1.6 C Language Access Example

```
void fndEn(FND_TypeDef* FNDx, uint32_t n) {
    FNDx->FCR = (n == 1) ? 0x01 : 0x00;
}

void fndfont(FND_TypeDef* FNDx, uint32_t fndFont) {
    FNDx->FDR = fndFont;
}

void fndDot(FND_TypeDef* FNDx, uint32_t Dot) {
    FNDx->FPR = Dot;
}

uint32_t fndBCD(FND_TypeDef* FNDx){
    return FNDx->BCD;
}
```

3.2 General Purpose I/Os (GPIO)

3.2.1 Introduction

각 범용 입출력 포트(GPIO)는 8비트 레지스터인 GPIOx_MODER(모드 설정), data register GPIOx_IDR(입력 데이터), GPIOx_ODR(출력 데이터)를 포함한다.

- I/O port control registers: 32-bit memory-mapped control registers (GPIOx_MODER) 는 I/O mode(input mode, output mode)를 선택하는데 사용된다.
- I/O port data registers: 8-bit memory-mapped data registers(input data register(GPIOx_IDR), output data register(GPIOx_ODR))

GPIOx_ODR은 출력할 데이터를 저장하고, 읽기/쓰기 모두 가능한 레지스터이며, GPIOx_IDR은 읽기 전용 레지스터이다.

3.2.2 GPIO main features

- 출력 데이터는 GPIOx_ODR 또는 Peripheral에서 제공됨
- 입력 데이터는 GPIOx_IDR 또는 Peripheral로 전달됨

3.2.3 GPIO registers

3.2.3.1 GPIO port mode register (GPIOx_MODER)

Address offset: 0x00

Bits 7:0 **MODE[7:0][0]**

해당 data는 I/O mode를 설정하기 위해 software에서 작성된다.

0: Input mode

1: output mode

3.2.3.2 GPIO port input data register (GPIOx_IDR)

Address offset: 0x04

Reset value: 0x000 00XX

Bits 7:0 IDR[7:0][0] 포트 핀의 입력 상태

해당 비트들은 read-only이며, 해당 GPIO 핀의 입력 상태를 나타낸다.

3.2.3.3 GPIO port output data register (GPIOx ODR)

Address offset: 0x08

Reset value: 0x0000 0000

31...8 7 6 5 4 3 2 1 0

Res	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
	rw							

Bits 7:0 **ODR[7:0][0]** 포트 핀의 출력 상태

해당 비트들은 software에 의해 읽고 쓸 수 있다.

3.2.3.4 GPIO register map

offset	Register	31...8	7	6	5	4	3	2	1	0
0000	MODER	Reserved								MODEy[7:0]
0004	IDR	Reserved								IDRy[7:0]
0008	ODR	Reserved								ODRy[7:0]

3.3 Ultrasound Peripheral

3.3.1 Introduction

Ultrasound Peripheral은 HC-SR04 센서와 데이터를 송수신하기 위해 설계한 모듈이다.

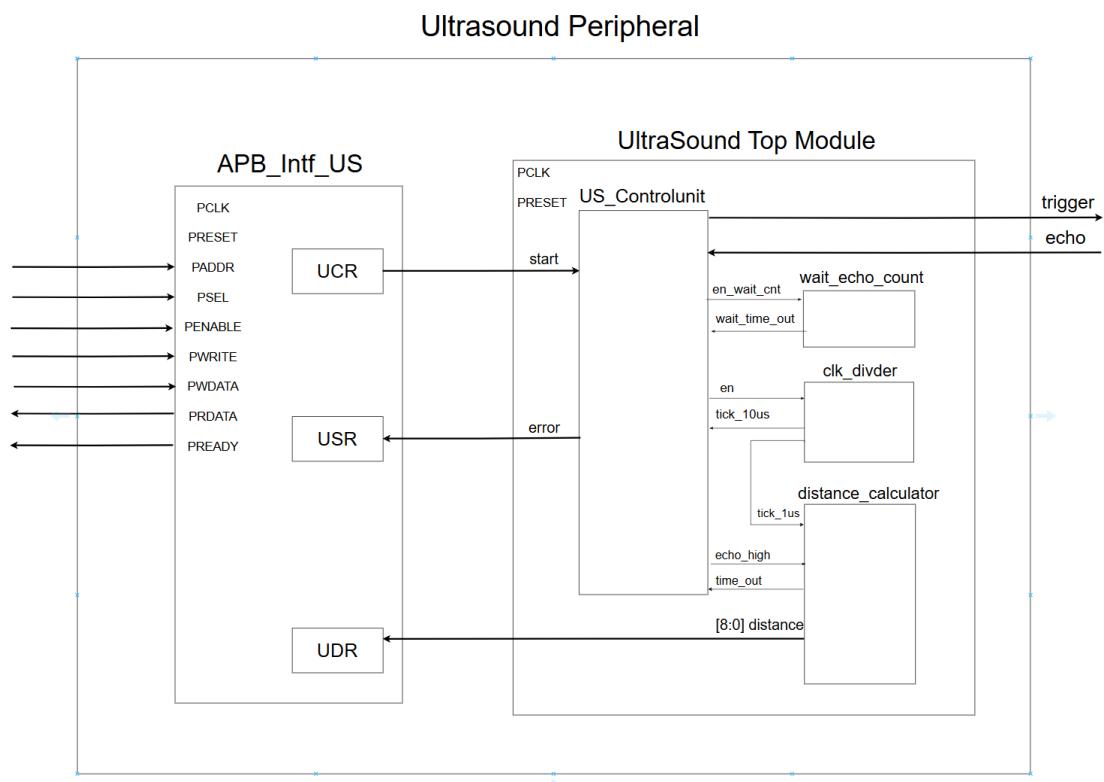
센서에 trigger 신호를 전달하여 측정을 시작하고, 측정 결과는 echo 신호로 수신되어 거리 값을 계산한다.

3.3.2 Main Features

- APB 프로토콜 기반의 초음파 거리 측정 Peripheral
- FSM을 기반으로 IDLE → TRIG → WAIT_ECHO → HIGH_ECHO 상태 전이 수행
- echo 상승/하강 에지 감지로 echo high 시간 측정
- echo high 시간은 1us 단위 tick으로 카운트하여 거리 계산
- 거리 계산: (echo high 시간(us) × 1130) >> 16으로 보정값 포함
- echo 미감지 또는 측정 시간 초과 시 error 상태 플래그 설정
- 모든 서브 블록은 동기적으로 PCLK과 PRESET으로 제어

- 거리 값은 최대 9비트(0~511cm)까지 지원 (UDR)

3.3.3 Functional Description



3.3.3.1 APB__intf_US 모듈

APB__intf_US 모듈은 APB Master로부터의 버스 신호를 수신하여 내부 제어 및 측정 결과 전달을 담당한다. 내부에는 다음과 같은 3개의 레지스터가 포함된다:

Offset	Register	31...9	8	7	6	5	4	3	2	1	0
0000	UCR										Start
0004	USR										Error
0008	UDR	Reserved									[8:0] distance

1. UCR (Ultrasound Control Register)

start 비트를 통해 측정을 트리거하며, 1cycle 후 자동 클리어됨

2. USR (Ultrasound Status Register)

echo 타임아웃 발생 시 error 플래그 설정

3. UDR (Ultrasound Data Register)

계산된 거리 값(cm 단위)을 저장하며 최대 511cm까지 표현 가능

3.3.3.2 Ultrasound_top 모듈

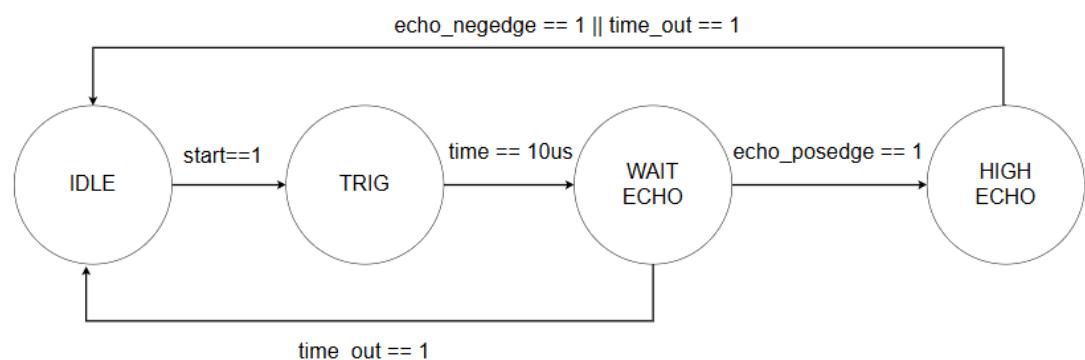
초음파 거리 측정의 핵심 로직을 담당하며, 다음과 같은 서브모듈로 구성된다:

1. US_Controlunit: FSM 기반 제어
2. clk_divider: 1us, 10us tick 생성
3. distance_calculator: echo high 시간을 카운트하고 이를 통해 거리 계산
4. wait_echo_counter: echo 입력 타임아웃 감지

3.3.4 설계 상세 내용

3.3.4.1 FSM 상세 동작 설명

초음파 센서의 동작 타이밍을 정확히 제어하기 위해 FSM 구조를 설계하였으며, 4개의 상태(IDLE, TRIG, WAIT_ECHO, HIGH_ECHO)로 구성된다.



State	Description	Transition Condition
IDLE	측정 요청 대기 상태	start_pulse 발생 시 (IDLE) → (TRIG)
TRIG	10us 트리거 펄스 출력	tick_10us 발생 시 (TRIG) → (WAIT_ECHO)
WAIT ECHO	echo 상승 에지 감지 및 대기	echo_rising 발생 시 (WAIT_ECHO) → (HIGH_ECHO) wait_time_out 발생 시 (WAIT_ECHO) → (IDLE)
HIGH ECHO	echo high 시간 측정 및 거리 계산	echo_falling 발생 시 (HIGH_ECHO) → (IDLE) time_out 발생 시 (HIGH_ECHO) → (IDLE)

FSM 동작에 대해 좀 더 상세하게 설명하면, 시스템은 초기 상태인 IDLE에서 외부로부터 start 신호가 유입되기를 대기하며 신호의 상승 에지가 감지되면 TRIG 상태로 전이하여 초음파 센서에 10us 간의 trigger 펄스를 출력한다.

10us가 경과하면 trigger 신호는 자동으로 비활성화되고 시스템은 WAIT_ECHO 상태로 전이하여 echo 신호의 상승 에지를 감지하게 된다. 만약 echo 신호가 5ms 이내에 감지되지 않으면 타임아웃이 발생하여 error 플래그가 설정되고 IDLE 상태로 복귀한다. 반대로 echo 상승 에지가 감지되면 HIGH_ECHO 상태로 진입하여 echo가 high 상태인 기간을 측정하고, 이 시간은 이후 거리 계산에 사용된다.

echo의 하강 에지가 감지되면 측정이 정상적으로 완료된 것으로 간주하여 IDLE 상태로 복귀하고, echo high 상태가 25ms 이상 지속되면 역시 타임아웃으로 판단하여 error 플래그를 설정한 후 IDLE로 돌아간다.

전체 FSM 흐름은 초음파 센서의 동작 특성을 안정적으로 제어하고, 오류 상황에서도 시스템이 일관되게 동작할 수 있도록 보장하기 위해 다음과 같이 설계하였다.

3.3.4.2 Echo 신호 동기적 검출

echo 신호는 외부 센서로부터 유입되는 비동기 신호이기 때문에 시스템 클럭 도메인 내에서 직접 처리할 경우 메타안정성 문제가 발생할 수 있다.

이를 방지하기 위해 echo 신호를 2단계 Flip-Flop으로 동기화하여 동기화된 신호 간의 상태 변화를 이용해 상승 엣지와 하강 엣지를 정확하게 검출한다.

이러한 방법을 통해 FSM의 상태 전이를 안정적으로 수행할 수 있도록 한다.

Systemverilog 코드 예시

```
logic echo_sync1, echo_sync2;

logic echo_rising, echo_falling;

assign echo_rising = !echo_sync2 & echo_sync1; // rising edge

assign echo_falling = echo_sync2 & !echo_sync1; // falling edge

always_ff @(posedge clk or posedge reset) begin

    if (reset) begin

        echo_sync1 <= 0;

        echo_sync2 <= 0;

    end else begin

        echo_sync1 <= echo;

        echo_sync2 <= echo_sync1;

    end

end
```

3.3.4.3 나눗셈 연산 최적화

거리 계산 방식은 하드웨어 구현 효율을 고려해 최적화되었다. 일반적인 방식에서는 echo high 시간을 us 단위로 측정한 후, 이를 58로 나누어 사용하지만, 나눗셈 연산은 FPGA에서 자원 소모가 크고 지연 시간이 길다는 단점이 있다.

이에 따라 나눗셈 대신, 고정 소수점 기반의 곱셈 및 시프트 연산을 활용해 거리 계산을 최적화 하였다. 구체적으로는 $(\text{echo_time} \times 1130) \gg 16$ 방식으로 연산하며 이는 $\text{echo_time} \div 58$ 과 거의 동일한 결과를 제공하면서도 곱셈과 시프트만으로 구현 가능해 연산 지연을 줄이고 DSP 블록 자원을 효율적으로 활용할 수 있다.

최종적으로 계산된 거리 값은 최대 9비트로 표현되며, APB 인터페이스를 통해 UDR(User Data Register) 레지스터에 전달된다.

리소스 유형	개선 전	개선 후	변화량	개선율
LUT	123	43	-80	-65%
Register	76	68	-8	-10.5%
DSP	0	1	+1	

3.3.5 Ultrasound Peripheral 검증

3.3.5.1 Introduction

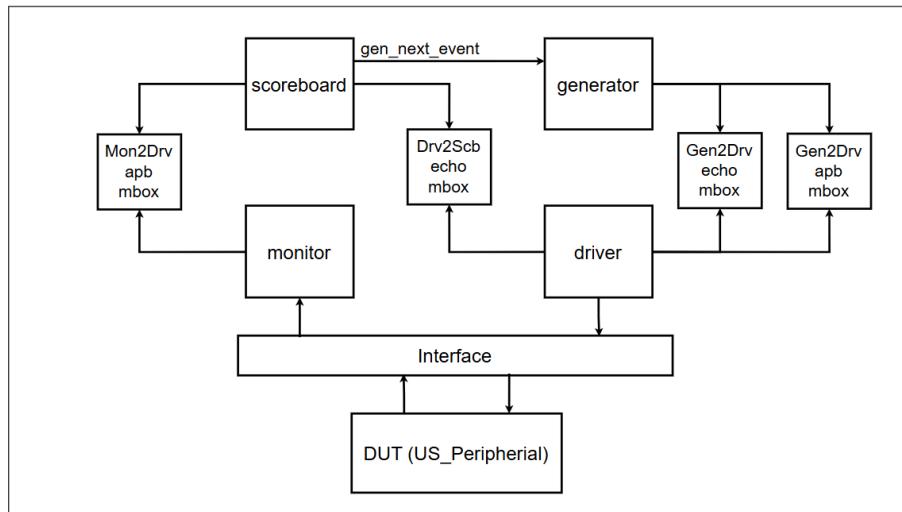
초음파 거리 측정 모듈(Ultrasound Peripheral)은 APB 버스를 통해 제어되며, 외부에서 발생한 echo 신호의 펄스 폭을 기반으로 거리 값을 계산하고 이를 레지스터에 저장하는 기능을 수행한다.

Testbench는 UVM 스타일의 구성 요소(generator, driver, monitor, scoreboard)를 기반으로 설계되었으며 각 구성 요소는 mailbox와 event를 통해 비동기적으로 연동되어 시나리오 기반의 기능적 검증이 가능하다.

또한 transaction_apb 및 transaction_echo 클래스를 기반으로 한 constrained random stimulus를 통해 정상 케이스와 다양한 타입의 시나리오(지연, 폭 초과)를 포함한 corner case까지 포괄하는 커버리지 기반 검증을 구현하였다.

아래는 Ultrasound Peripheral 검증을 위한 전체 테스트벤치 블록 다이어그램을 보여준다

Ultrasound Systemverilog Testbench



3.3.5.2 Testbench Structure

Ultrasound Peripheral 검증 환경은 SystemVerilog OOP 기반의 클래스를 활용한 유사-UVM 스타일의 구조로 구성되었다. 각 구성 요소는 모듈 간의 결합도를 낮추고 재사용성을 높이기 위해 mailbox 및 interface 기반의 연결을 사용하며 다음과 같은 구성 요소들로 이루어져 있다.

구성 요소	설명
Transaction_apb	APB 트랜잭션 정보를 포함한다. PADDR,PWRITE,PSEL 등을 포함하며 randomize() 기반의 시나리오 생성
Transaction_echo	Echo 지연시간, 펄스 폭, 타임아웃 여부 등을 제약 조건 기반으로 랜덤 생성
Generator	APB 및 echo 트랜잭션을 생성하여 driver에 전달. 이후 gen_next_event 이벤트를 수신할 때까지 대기.
Driver	DUT 인터페이스에 APB 트랜잭션 값을 전달. APB protocol에 맞게 신호 값을 전송하며, trigger 발생일 경우 (UCR에 1 write일 경우) echo신호 또한 전달.
Monitor	PREADY =1 이 발생할 때 인터페이스로부터 값을 읽어서 내부 APB 트랜잭션에 저장하고 이를 Scoreboard로 전달
Scoreboard	예상 결과(echo 트랜잭션 내 값들)와 실제 DUT의 PRDATA값을 비교하여 PASS/FAIL 판별. 판별이 끝나면 gen_next_event를 발생시켜 generator가 다음 트랜잭션을 생성하도록 함.
Environment	전체 테스트벤치 구성요소들을 인스턴스하고 fork join 구문을 활용하여 병렬적으로 실행.
Mailbox	apb mbox는 transaction apb 값을 전달, echo mbox는 transaction echo 값을 전달.

3.3.5.3 Test Scenarios

본 테스트벤치는 다양한 실제 상황을 반영하기 위해 랜덤 제약 기반의 시나리오를 자동 생성하여 다음과 같은 경우의 수를 검증한다.

테스트 항목	설명
정상 동작	Tigger =1 시 echo pulse가 정상적으로 들어오고, 거리에 해당하는 값이 UDR에 저장됨
Delay timeout	ehco가 들어오기 까지 시간이 너무 길어 타임아웃 발생
Width Timeout	ehco의 pulse 폭이 너무 길어 타임아웃 발생 (센서가 측정할 수 있는 거리값을 초과)
Trigger 미입력	Trigger = 0 일때 측정 없이 정상적으로 무반응 처리되어야 함
잘못된 접근	읽기 전용 레지스터에 쓰기를 시도하거나, 쓰기 전용 레지스터를 읽는 경우

위와 같이 다양한 테스트 케이스를 동작시키기 위해 transaction_apb와 transaction_echo에서 각각 제약조건 기반으로 값을 생성한다.

transaction_apb는 APB 접근 시나리오를 제약 조건으로 구성하며, UCR 레지스터는 전체 접근의 60%로 가장 많이 사용되고, USR과 UDR은 각각 20% 비율로 설정된다. UCR은 쓰기 중심(90%)으로, trigger 값을 1 또는 0으로 설정하며, USR과 UDR은 읽기 중심(90%)으로 제약되어 있다. 잘못된 접근을 유도하기 위해 읽기 전용 레지스터에 쓰기 시도나 쓰기 전용 레지스터에 대한 읽기도 일부 확률로 포함된다.

transaction_echo는 echo 신호의 타이밍 및 상태를 제약하며, 전체의 80%는 정상 동작으로 echo 가 적절한 시간과 폭을 갖도록 설정되고, 나머지 20%는 timeout 시나리오로 구성된다. timeout 발생 시에는 delay timeout과 width timeout을 각각 50% 비율로 선택하고, 각 유형에 따라 delay 나 width의 범위를 비정상적으로 크게 설정한다. 정상일 경우 echo 폭을 기반으로 거리값을 계산 하며, timeout일 경우 거리는 0으로 처리된다.

3.3.5.4 Testbench Results & Analysis

테스트벤치를 10000회 실행한 결과, 다음과 같은 통계 데이터를 기반으로 DUT의 정확성과 안정성을 확인하였다.

```

==          Final Report          ==
=====
-----Valid Access Test-----: 8975
1. Write Test (UCR Trigger)   : 1533
  1-1. Timeout case          : 381
  1-2. Normal case           : 358
  1-3. No Trigger case       : 794
2. Read Test (USR + UDR)     : 7442
  2-1. USR (Status) Read     : 1157
  2-2. UDR (Distance) Read   : 6285
  2-3. No measure data       : 0
3. PASS  Test                : 8975
4. FAIL  Test                : 0

-----Invalid Access Test-----: 1025
5. UCR Read (Invalid)        : 172
6. USR Write (Invalid)       : 148
7. UDR Write (Invalid)       : 705

8. Total Test                 : 10000
=====
==      test bench is finished!  ==
=====
```

1. Valid Access Test (8975건)

- UCR Write (Trigger 관련): 1,533 건
 - Timeout case: 381 (25%)
 - 정상 동작: 358 (23.4%)
 - No Trigger: 794 (51.8%)
- USR/UDR Read (측정값 확인): 7,442 건
 - USR (Status) Read: 1,157 건
 - UDR (Distance) Read: 6,285 건
 - No measure data: 0 건

2. Invalid Access Test (1025건)

- UCR 읽기 시도: 172 건 (UCR 은 write 전용)
- USR 쓰기 시도: 148 건 (USR 은 read 전용)
- UDR 쓰기 시도: 705 건 (UDR 은 read 전용)

해당 테스트 결과는 초음파 Peripheral모듈이 정상적인 거리 측정은 물론, echo 지연 및 pulse 폭

에 따른 타임아웃 상황, 그리고 잘못된 레지스터 접근과 같은 다양한 예외 시나리오에 이르기까지 모든 상황에서 기대한 동작을 안정적으로 수행함을 보여준다.

3.4 Universal Asynchronous Receiver Transmitter (UART)

3.4.1 UART introduction

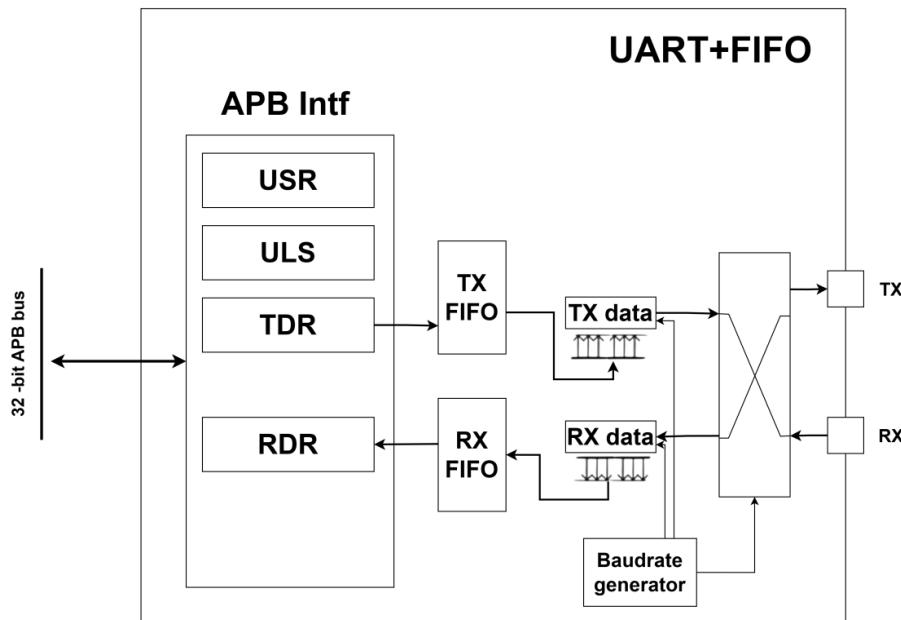
3.4.2 UART main features

UART features	UART
UART data length	8 bits
Tx/Rx FIFO	Supported
Tx/Rx FIFO size	8
baud rate	9600bps

Full Duplex Operation (Independent Serial Receive and Transmit Registers)

3.4.3 UART functional description

3.4.3.1 UART block diagram



3.4.3.2 UART signal

UART bidirectional communications

UART 양방향 통신에는 최소 2개의 Pin이 필요: Receive Data In (RX), Transmit Data Out(TX)

RX (Receive Data Input)

RX는 Serial Data input

TX (Transmit Data Output)

TX가 data를 전송할 필요가 없는 경우, TX pin은 High로 유지된다.

Table xx. UART input/output pins

Pin name	Signal type	Description
rx	Input	Serial data 수신 핀
Tx	Output	Serial data 송신 핀

3.4.3.3 UART FIFO

UART는 Transmit FIFO (TXFIFO)와 Receive FIFO(RXFIFO)가 제공된다.

Byte 단위로 data를 수신, 송신하기 위해 FIFO TX, RX의 data length를 8bit로 설정하였다.

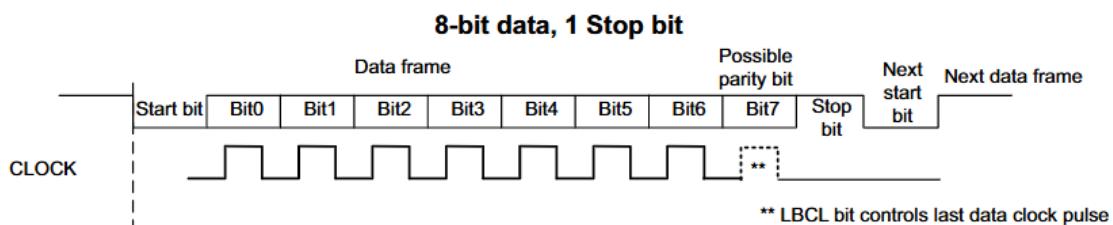
UART FIFO의 status는 USR register에 저장된다.

3.4.3.4 UART transmitter

Transmitter는 1 byte character 을 전송할 수 있도록 설계되었다.

Transmit data register(TDR)에 전송할 데이터가 저장된다.

데이터 전송 시작 시 start bit (low logic level)이 one bit period 동안 유지되며, 데이터 전송이 끝나면 stop bit(high logic level)이 one bit period 동안 유지된다.



Character transmission procedure

Character 송신을 위해 TDR register에 write data를 send 하면 TXFNF (TXFIFO Not Full) flag 가 high일 때 Write data가 TXFIFO에 push 된다. TXFNF flag는 TXFIFO 상태에 따라 하드웨어에서 자동적으로 정의된다.

C Code Example

```
void UART_writeData (UART_TypeDef *UARTx, uint32_t data) {  
    while((UART_state(UARTx) && (1 << 1)));  
    UARTx -> TDR = data;  
}
```

위 함수는 TXFIFO가 full인 경우 full이 아닐 때 까지 wait 한다.

3.4.3.5 UART receiver

Receiver는 1 byte character 을 수신할 수 있도록 설계되었다.

Character reception

Character 수신 후 RXFNE (RXFIFO Not Empty) flag가 high 이면 RXFIFO에 read byte가 저장된다.

RXFNF flag는 RXFIFO 상태에 따라 하드웨어에서 자동적으로 정의된다.

C Code Example

```
void UART_writeData (UART_TypeDef *UARTx) {  
    while((UART_state(UARTx) && (1 << 0)));  
    return UARTx-> RDR;  
}
```

Overrun error

Rx 송신이 완료되고 rx 처리된? 데이터를 FIFO로 전달할 준비가 되었을 때, FIFO가 가득찬 경우 발생한다. 이 경우 RX FIFO에 하나의 빈공간이 생기기 전까지 shift register에서 RDR register로 데이터를 전달할 수 없다.

Rx fifo 가 data를 전달할 때까지 ~~~ (그 push<- pop 설명넣기)

3.4.4 UART baud rate generation

Receiver와 Transmitter의 Baud rate는 9600bps로 설정하였으며, 16번 oversampling 하였다.

$$\text{Tx/Rx baud} = \frac{100\text{MHz}}{9600 \times 16}$$

3.4.5 UART register

3.4.5.1 UART Status Register (USR)

Address offset: 0x00

Reset value: 0x0000 0000

31...4	3	2	1	0
Res	RXFF	TXFE	TXFF	RXFE
	r	r	r	r

Bit 3 **RXFF**: RXFIFO Full

received data의 개수가 FIFO size 와 같을 때 high로 hardware로부터 설정된다.

0: RX FIFO not full

1: RX FIFO full

Bit 2 **TXFE**: TXFIFO Empty

TXFIFO가 empty 일 때 set되며 FIFO가 하나의 data라도 포함하는 순간 flag는 지워진다

0: TX FIFO is not empty

1: TX FIFO is empty

Bit 1 **TXFF**: TXFIFO Full

TXFIFO가 가득 차 있는 경우 hardware에 의해 setting되며 TDR register에 write 가능하다. TDR에 대한 모든 write operation 값은 TXFIFO에 put된다. 해당 flag는 FIFO가 full이 되기까지 유지되며, TXFIFO가 full이 되면 UDR 같은 TXFIFO에 write될 수 없다.

0: TX FIFO is not full

1: TX FIFO is full

Bit 0 **RXFE**: RXFIFO Empty

RXFIFO가 비어 있다면 hardware에 의해 1로 setting 되며, RDR register를 통해 RXFIFO값

을 읽어올 수 있음을 의미한다.

0: Received data is ready to read

1: Data is not received

3.4.5.2 UART transmit data register (TDR)

Address offset: 0x08

Reset value: 0x0000 0000

	31...8	7	6	5	4	3	2	1	0
Res	TDR[7:0]								
	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:8 Reserved, must be kept at reset value

Bits 7:0 **TDR[7:0]**: Transmit data value

Note: 해당 Register는 TXFF가 0일 때만 write 가능하다.

3.4.5.3 UART receive data register (RDR)

Address offset: 0x0C

Reset value: 0x0000 0000

	31...8	7	6	5	4	3	2	1	0
Res	RDR[7:0]								
	r	r	r	r	r	r	r	r	r

Bits 31:8 Reserved, must be kept at reset value

Bits 7:0 **RDR[7:0]**: Receive data value

3.4.5.4 UART register map

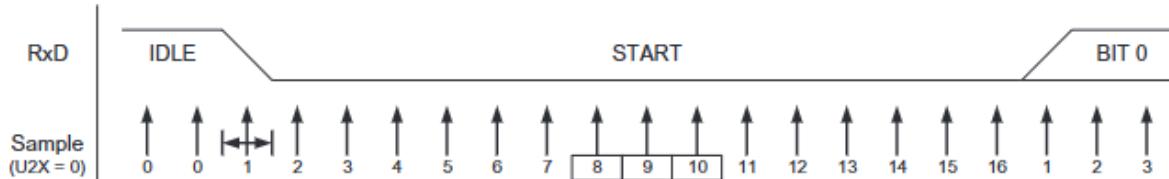
offset	Register	31...8	7	6	5	4	3	2	1	0
0000	USR						RXFF	TXFE	TXFNF	RXFNE
0004	ULS									ULS
0008	TDR	Reserved					TDR[7:0]			
000C	RDR	Reserved					RDR[7:0]			

3.4.6 Asynchronous Data Reception

UART 통신은 송신 측과 수신 측이 공동 클럭을 공유하지 않는 대신, 같은 baud rate로 데이터를 주고 받는다. 하지만 같은 baud rate로 설정되었다고 해도 수신 측에서 어느 시점에 데이터를 샘플링 하느냐에 따라 오차가 크게 달라진다.

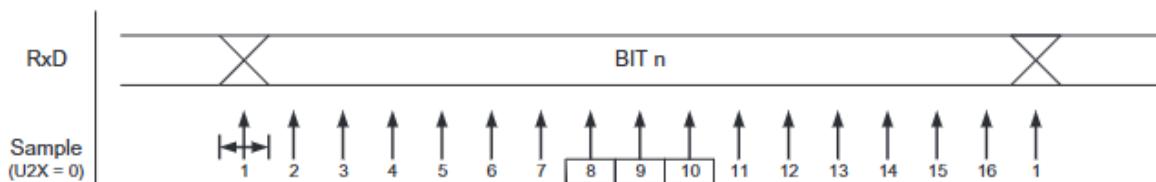
설계한 송/수신 모듈은 9600 bps로 설정되었고, 수신 측은 이의 16배로 oversampling 하도록 설계하였다.

3.4.6.1 Start bit sampling



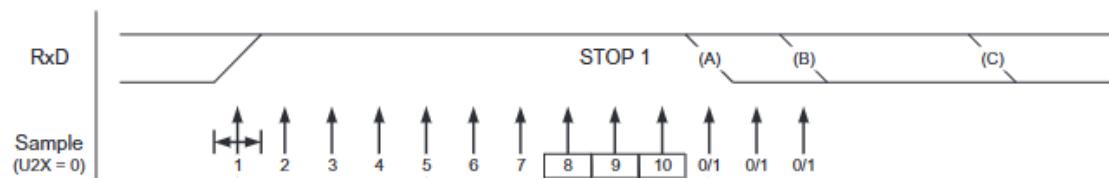
rx line에서 falling edge를 감지하면, 그 순간의 첫번째 sample을 1이라고 가정한다. 이 sample부터 $\frac{100MHz}{9600 \times 16}$ 간격으로 sampling을 이어가며 8번째 sample이 0이라면 start state로 판단한 후 data state로 전이한다.

3.4.6.2 Data bit sampling



Character 수신을 위해 각 data bit는 16개의 샘플 중 8번째 sample을 sampling 하며, 이 과정은 총 8bit에 대해 수행된다.

3.4.6.3 Stop bit sampling



Stop state 시작부터 16 oversampling 이후 수신 작업을 끝내며, 16 oversampling 이후에 들어온 falling edge에 대해 수신을 재기한다.

4. Firmware Implementation

4.1 Introduction

본 펌웨어는 RV32I 기반 멀티사이클 MCU에서 실행되며, APB 버스를 통해 연결된 주변장치(UART, 초음파 센서, 온습도 센서, FND, 버튼, 스위치)를 직접 제어하여 센서 데이터 수집, 시각화 출력 및 통신 기능을 통합적으로 수행한다.

프로젝트의 명칭은 '실시간 환경 모니터링 시스템'으로, 초음파(HC-SR04) 및 온습도(DHT11) 센서를 수동 모드와 자동 모드 두 가지 방식으로 제어할 수 있도록 설계되었다.

수동 모드에서는 버튼 입력을 통해 센서를 트리거하며, 자동 모드에서는 UART 명령 입력을 기반으로 FSM상태를 전이시키고, 각 상태에 따라 주기적인 센서 측정 함수를 실행한다.

센서 측정 결과는 FND를 통해 실시간으로 출력되며, UART를 통해 ASCII 형식으로 송출된다. 스위치 입력에 따라 FND 출력 대상(거리, 온도, 습도)을 선택할 수 있다.

4.2 시스템 구성

구성요소	설명
MCU	RV32I 기반 멀티사이클 MCU
Bus 구조	AMBA APB 프로토콜 기반 Peripheral 인터페이스
센서 인터페이스	초음파(HC-SR04), 온습도(DHT11)
출력 장치	FND (7-segment), UART (ASCII)
입력 장치	Button, Switch

4.3 동작 시나리오

4.3.1 수동 측정 모드 (Manual Mode)

- 초음파: BTNL 입력 시 측정
- 온습도: BTNR 입력 시 측정

4.3.2 자동 측정 모드 (FSM 기반)

- UART 'u' 입력 → 초음파 모드 진입
'1'~'3' 입력에 따라 주기적으로 거리 측정 수행 (측정 주기 선택 가능)
- UART 't' 입력 → 온습도 모드 진입
'4' or '5' 입력에 따라 주기적으로 온습도 측정 수행 (측정 주기 선택 가능)

4.3.3 FND 출력 선택

- SW0 == 0 → FND 에 초음파 거리 표시
- SW0 == 1 && SW1 == 0 → FND 에 온도 표시
- SW0 == 1 && SW1 == 1 → FND 에 습도 표시

4.4 주요 코드 및 제어 로직

4.4.1 FSM 기반 UART 명령 처리

(1) 동작 설명

UART를 통해 수신되는 특정 명령에 따라 시스템의 상태를 전이시키는 **유한 상태 머신(FSM)** 구조를 구현하였다. 사용자로부터 'u', '1', '2', '3', 'x' 등의 명령을 수신하여 상태 이동이 일어나고, 각 상태에 맞는 센서 측정 동작을 수행한다.

예를 들어, 'u'를 수신하면 초음파 센서 관련 대기 상태(US_READY)로 진입하고, 이 상태에서 '1'을 수신하면 첫 번째 센서 측정 함수(FUNC1)를 실행하는 방식이다.

(2) 활용 문법 및 개념

- switch-case 문: 현재 FSM 상태에 따라 코드 실행 경로를 효율적으로 분기.
- enum: 상태의 가독성을 높이고 잠재적인 오류를 줄이기 위해 IDLE, US_READY, TEMP_READY, FUNC1 등으로 각 상태를 명시적으로 정의.

(3) 코드 내용

```
// FSM state
enum {IDLE, US_READY, TEMP_READY, FUNC1, FUNC2, FUNC3, FUNC4, FUNC5, FUNC6};

// main 함수 내부의 FSM
// 초음파 FSM (func1~3)
switch (state) {
    case IDLE:
        if (uart_rdata_us == 'u' || uart_rdata_us == 'U') state = US_READY;
        break;

    case US_READY:
        if (uart_rdata_us == '1') state = FUNC1;
        else if (uart_rdata_us == '2') state = FUNC2;
        else if (uart_rdata_us == '3') state = FUNC3;
        else if (uart_rdata_us == 'x' || uart_rdata_us == 'X') state = IDLE;
        break;

    case FUNC1:
        func1(&func1PrevTime, &func1Data, &us_dist);
        if (uart_rdata_us == '2') state = FUNC2;
        else if (uart_rdata_us == '3') state = FUNC3;
        else if (uart_rdata_us == 'x' || uart_rdata_us == 'X') state = IDLE;
        break;

    case FUNC2:
        func2(&func2PrevTime, &func2Data, &us_dist);
        if (uart_rdata_us == '1') state = FUNC1;
        else if (uart_rdata_us == '3') state = FUNC3;
        else if (uart_rdata_us == 'x' || uart_rdata_us == 'X') state = IDLE;
        break;
}
```

```

break;

case FUNC3:
    func3(&func3PrevTime, &func3Data, &us_dist);
    if (uart_rdata_us == '1') state = FUNC1;
    else if (uart_rdata_us == '2') state = FUNC2;
    else if (uart_rdata_us == 'x' || uart_rdata_us == 'X') state = IDLE;
    break;
}

// 온습도 FSM (func4, func5)

switch (state_t) {
    case IDLE:
        if (uart_rdata_temp == 't' || uart_rdata_temp == 'T') state_t = TEMP_READY;
        break;

    case TEMP_READY:
        if (uart_rdata_temp == '4') state_t = FUNC4;
        else if (uart_rdata_temp == '5') state_t = FUNC5;
        else if (uart_rdata_temp == 's' || uart_rdata_temp == 'S') state_t = IDLE;
        break;

    case FUNC4:
        func4(&func4PrevTime, &func4Data, &dht_t, &dht_h);
        if (uart_rdata_temp == '5') state_t = FUNC5;
        else if (uart_rdata_temp == 's' || uart_rdata_temp == 'S') state_t = IDLE;
        break;

    case FUNC5:
        func5(&func5PrevTime, &func5Data, &dht_t, &dht_h);
        if (uart_rdata_temp == '4') state_t = FUNC4;
        else if (uart_rdata_temp == 's' || uart_rdata_temp == 'S') state_t = IDLE;
        break;
}

```

4.4.2 타이머 모듈을 활용한 주기적 센서 측정

(1) 동작 설명

타이머 모듈을 활용하여 FSM의 각 상태에서 주기적으로 센서 데이터를 측정할 수 있도록 구현하였다.

현재 TIMER의 현재 카운트 값(TCNT)과 이전 측정 시점(prevTime)을 비교함으로써 타이머 모듈 하나를 사용하여 여러 시간 주기를 측정할 수 있다.

이러한 방식은 비차단식 방식으로 타이머 인터럽트 없이도 다양한 주기의 동작을 구현할 수 있다.

(2) 코드 내용

```

void func1(uint32_t *prevTime, ...) {
    uint32_t curTime = TIM_readCount(TIM);

```

```
if (curTime - *prevTime < 500) return;  
  
*prevTime = curTime;  
  
// 주기 동작 (센서 측정, UART 전송 등)  
  
}
```

타이머의 현재 값과 이전에 저장한 값의 카운트 차이를 비교하여 설정된 주기마다 동작을 수행하는 구조이다. (현재 코드는 500ms)

코드 하단에 센서 측정, 데이터 전송 등의 작업을 넣으면 해당 주기마다 반복 실행된다.