



C – HW5

임베디드스쿨1기

Lv1과정

2020. 08. 28

박성환

1. Review(CISC vs RISC)

1. CISC

- **CISC는 파이프라이닝을 아예 사용 못한다?**

⇒ CISC는 파이프라이닝에 최적화 되어있지는 않으나 CPU 내부적으로는 Cache 메모리를 통해 복잡한 명령어를 다시 단순한 명령들로 나누어 명령어 파이프라인에서 처리하기 때문에 실제 내부 작동원리는 RISC와 같다고 할 수 있음

- **CISC는 폰노이만 구조인가?**

⇒ CPU와 주메모리 사이의 관계는 폰노이만 구조지만 내부에 Instruction Cache, Data Cache를 둬으로써 Core와 Cache 사이에는 하버드 구조를 택하고 있음

- **왜 CISC는 파이프라이닝에 최적화 되어있지 않다고 말할까?**

⇒ 우선 명령어 사이즈가 고정인 RISC와 달리 가변적이고 명령어마다 잡아먹는 실행 Cycle이 다르기 때문에 최적화에 어려움이 있음

2. RISC

- **파이프라이닝에 왜 최적화 되어있나?**

⇒ 고정 길이의 명령어를 사용하며 모든 연산은 하나의 클럭으로 실행되므로 레지스터뿐만 아니라 메모리에서 연산을 수행할 수 있는 명령어 세트 구조

- **RISC의 장점을 한마디로 말하면?**

=> 간단한 고정된 명령어로 파이프라이닝 최적화를 통해 빠른 실행 속도를 보임

1. Review(CISC vs RISC)

1. Load & Store Architecture

- Load(메모리 to 레지스터)명령과 Store(레지스터 to 메모리)명령만 메모리 액세스 가능하도록 설계된 구조
- 예를 들어 덧셈 연산을 하더라도 메모리끼리 바로 더하는 것이 안되고 Load를 통해 레지스터에 값들을 불러와서 더한 다음에 결과를 다시 레지스터에 저장하고 이를 Store를 통해 메모리에 저장하는 식의 구조

예) ADD 연산에서 피연산자가 하나는 레지스터 하나는 메모리에 있는 것은 절대 가능하지 않음

2. Register Memory Architecture

- 레지스터뿐만 아니라 메모리에서 연산을 수행할 수 있는 명령어 세트 구조

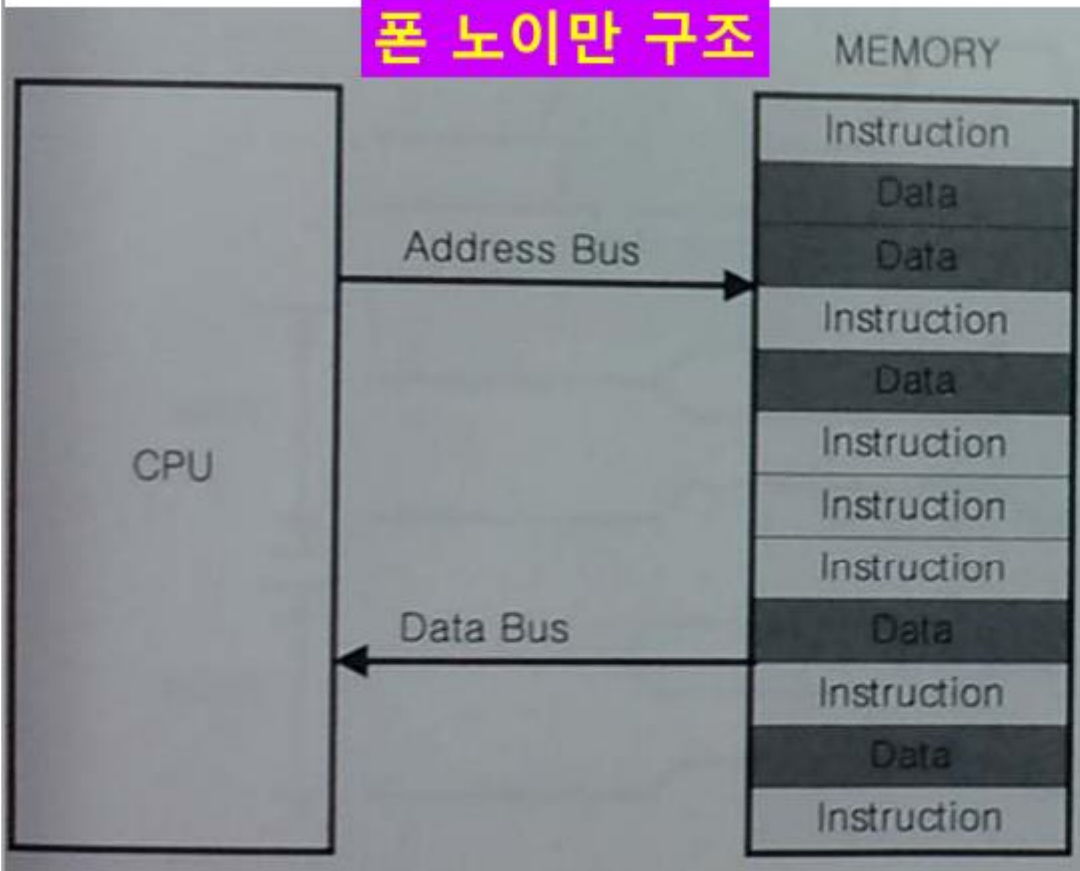
예) ADD 연산에서 모든 피연산자가 메모리나 레지스터에 있을 수 있음

3. RISC(Load & Store) vs CISC(Register Memory)

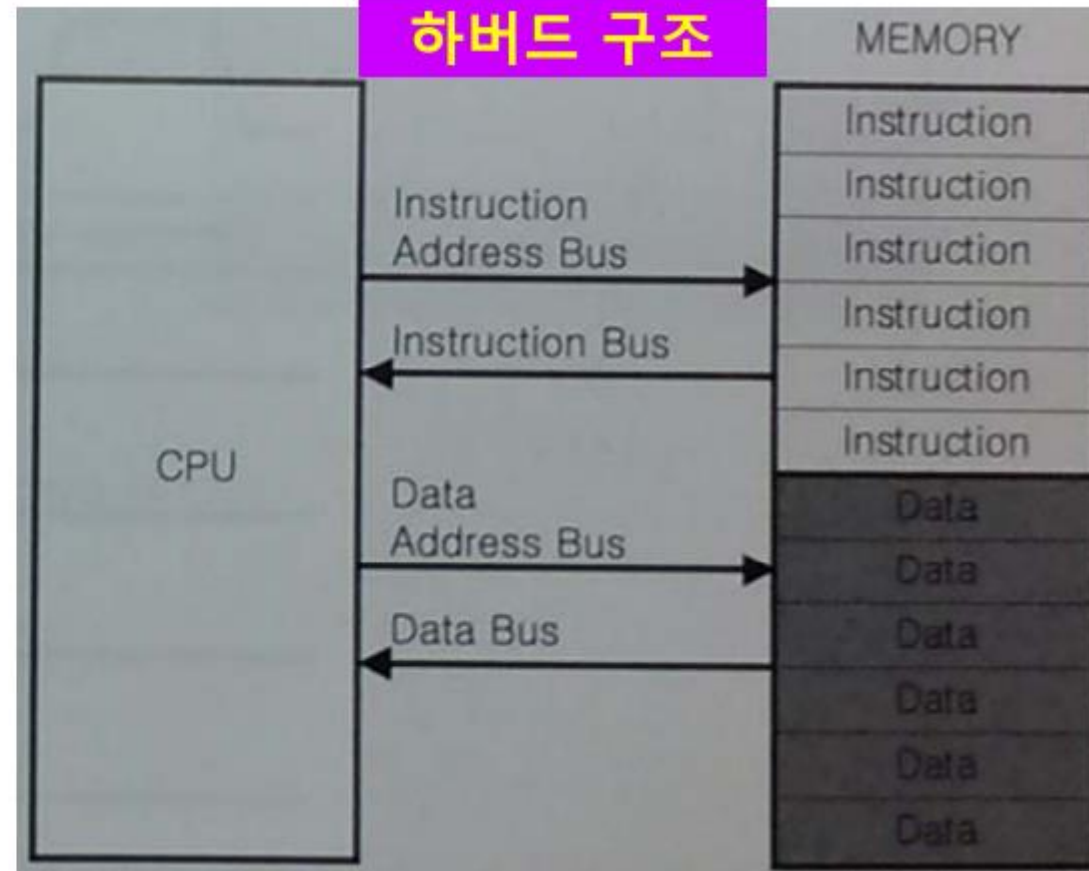
- 일단 기본적인 RISC 아키텍처는 메모리 2 메모리 연산이 불가능하다.
그렇기 때문에 Load 이후에 Store를 통해서 메모리를 제어한다.
⇒ 레지스터 뱅크를 많이 두었고 레지스터 사이의 연산만 실행하여 불필요한 메모리 접근을 줄임
- 반면 CISC 아키텍처는 메모리 2 메모리 연산이 가능하다.
그렇기 때문에 mov가 처리할 수 있는 케이스가 여러가지가 된다.
레지스터 2 메모리, 메모리 2 레지스터, 레지스터 2 레지스터, 메모리 2 메모리

참고. 폰노이만 vs 하버드

폰 노이만 구조

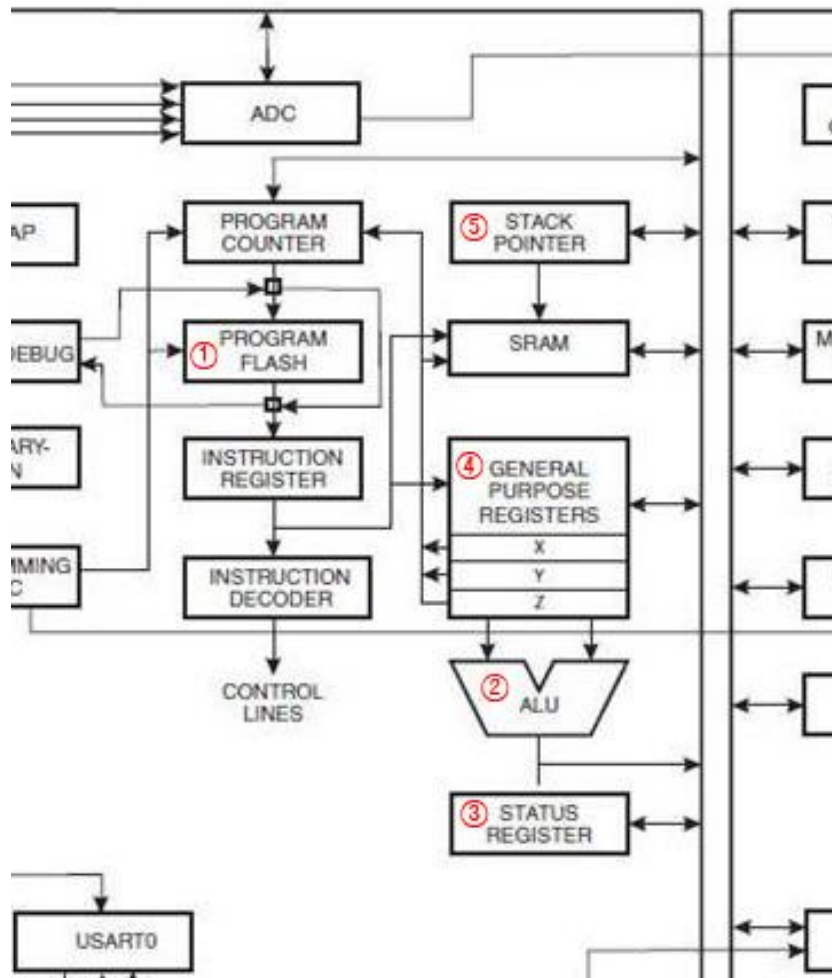


하버드 구조



2. AVR구조

1. AVR Core 구조(1)



① Program Flash Memory

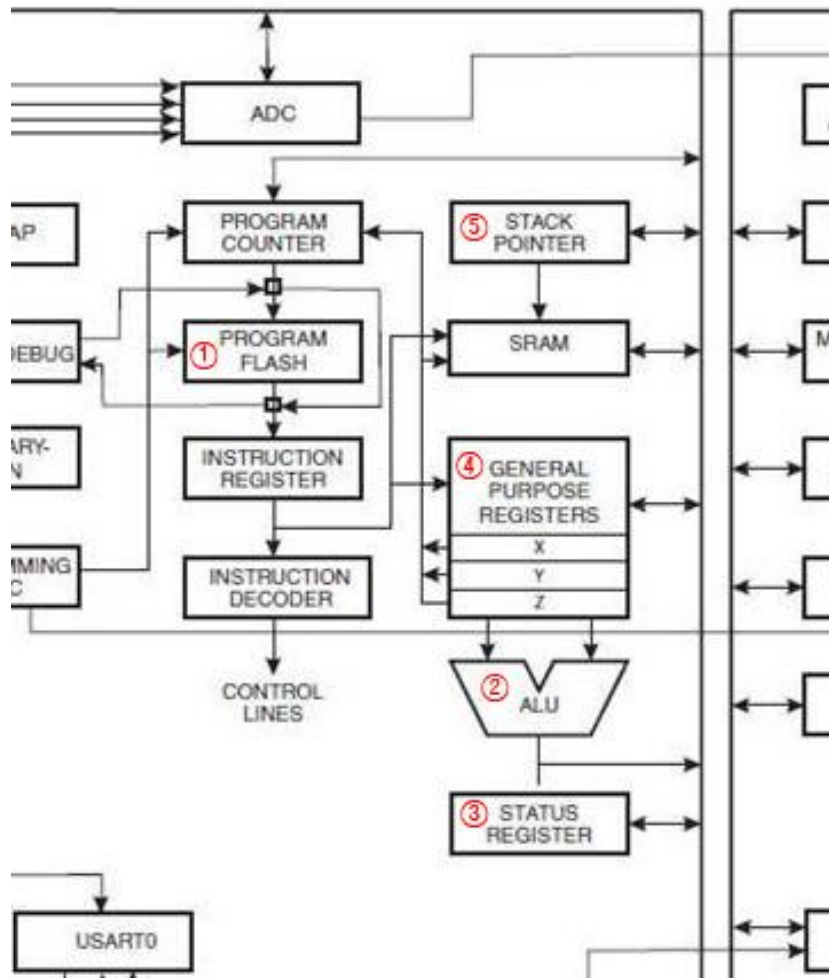
프로그램 메모리는 8비트로 구성되어있지만 기본적으로 한 개의 번지가 16비트 단위로 구성되어 16비트 마이크로프로세서인 것처럼 동작함(8bit = 1byte / 16bit = 2byte)

② ALU(산술-논리 연산자)

- 32개 general purpose register와 직접적으로 연계되어 동작함.
- 즉, ALU는 두뇌의 산술처리하는 부분에 해당되며 32개의 목적레지스터는 두뇌의 순간기억장치에 해당
- 이 두개가 서로 연계되어 더하기 빼기 등을 하며 동작하는 것입니다.
- 레지스터 간 또는 레지스터와 상수 간의 산술 또는 논리 연산을 단일 클럭 사이클에 수행 연산된 결과에 대한 ALU의 상태를 상태 레지스터로 갱신 됨(3번 상태레지스터참고)
- 강력한 하드웨어 곱셈기를 가지고 있어서 부호있는 정수/부호없는 정수의 곱셈 연산과 소수점 형식의 곱셈 연산을 빠르게 수행할 수 있음.
- 레지스터간 연산은 보통 1사이클 명령으로 끝남

2. AVR구조

1. AVR Core 구조(2)



③ 상태 레지스터(Status Register)

- 가장 최근에 실행된 산술 연산의 명령어 처리 결과 에 대한 상태를 나타내 주는 레지스터
- 조건부 처리 명령에 의해 프로그램의 흐름을 변경하는데 사용될 수 있음(조건부 명령이란 FOR,WHILE,SWITCH.. 등을 말함)
- 인터럽트를 처리하는 과정에서 자동으로 저장되거나 복구되지 않으므로, 반드시 소프트웨어에서 이러한 동작을 처리하여 주어야 함

Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

I(7) : interrupt enable 1 = 전체 interrupt enable, 0 = 전체 인터럽트 disable
(주로 코딩을 하시면 SREG 레지스트 중 i비트를 가장 많이 다룸.)

T(6) : 비트 복사 저장, T bit를 통하여 bit 전송
(UART통신시 사용하면 편합니다)

H(5) : half carry flag : 0000 1000 + 0000 1000 => half carry 발생

S(4) : sign bit : $V \text{ XOR } N$ (V_{MSB} , N_{MSB})

V(3) : **overflow bit** : 1000 0000 + 1000 0000 => **overflow bit set**
(오버플로는 음수+음수 = 양수 일때 발생합니다)

N(2): negative bit : 연산결과가 음수임

Z(1): zero bit : 연산결과가 0임을 나타냄

C(0): carry bit : 연산결과 자리 수 올림(더하기), 혹은 빌림(빼기)

위의 상태레지스터는 i비트빠고 잘 사용하지 않지만 후에 코딩이 길어지거나 프로그램의 조건문이 너무 길어지거나 애매할때 조금만 고민해보시면 사용이 편하다는것을 파악 가능

2. AVR구조

1. AVR Core 구조(3)

b7	b6	b5	b4	b3	b2	b1	b0	addr.
R0								\$00
R1								\$01
⋮								⋮
R14								\$0E
R15								\$0F
R16								\$10
R17								\$11
⋮								⋮
R24								\$18
R25								\$19
R26 (XL)								\$1A X레지스터 하위바이트
R27 (XH)								\$1B X레지스터 상위바이트
R28 (YL)								\$1C Y레지스터 하위바이트
R29 (YH)								\$1D Y레지스터 상위바이트
R30 (ZL)								\$1E Z레지스터 하위바이트
R31 (ZH)								\$1F Z레지스터 상위바이트

1Byte

4

범용 레지스터

ALU가 산술/논리 연산을 할 때 피연산자를 여기서 일경오며
연산결과도 여기로 저장

예) $R0 = R0 + R1$ ($R0, R1$ 에는 add 연산의 피연산자들)

- 어셈블러에서 아래와 같이 참조

0x00 – 0x1F 같은 주소로 참조(x)

R0 – R31 이름으로 참조(o)

- R0-R25 : 데이터 저장 및 연산에 쓰임

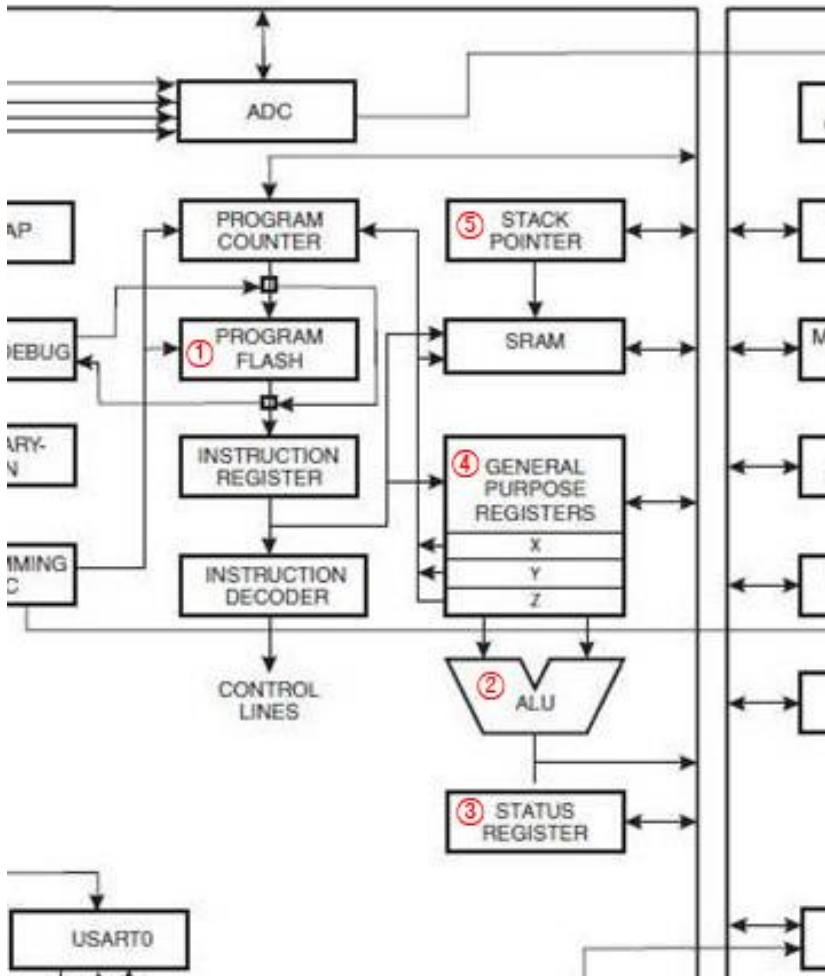
상위 R0 – R15 : 직접 상수 대입 불가

하위 R16 – R31 : 직접 상수 대입 가능

- R24 – R31 : 2개씩 묶어서 16bit 단위로 연산을 하기 용이하며
각각 X, Y, Z레지스터로 명명하여 16bit 연산 뿐만 아니라
주소를 지정하거나 주소 연산을 하는데도 사용

2. AVR구조

1. AVR Core 구조(4)



5 스택포인터(SP)

Bit	15	14	13	12	11	10	9	8	SPH
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

- 서브루틴이나 인터럽트 발생시에 복귀되는 주소를 임시로 기억하기 위해서 사용되거나 일반 프로그램에서 지역 변수 또는 임시 데이터를 저장하는 용도로 사용되는 LIFO(Last In First Out) 구조의 메모리
- 스택 포인터는 항상 데이터의 상단(top of stack)을 가리키는 **16비트 레지스터**로서 SP라고 표시되는데, 이는 데이터 저장 가능한 스택의 번지를 의미하는것
- 스택은 SRAM 영역내에 존재, SP 레지스터의 초기값은 적어도 **0x60** 번지 이상의 값으로 설정

참고. RAMPZ(RAM Page Z Select Register)

RAMPZ 레지스터란?

- AMPZ 레지스터는 Z 포인터에 의해 접근 가능한 64k RAM Page를 선택할 때 사용됨
- ATmega128은 64k바이트 이상의 외부 메모리를 지원하지 않기 때문에 LPM/SPM 명령어가 사용될 때, 프로그램 메모리의 어느 페이지에 접근할지를 선택할 때 사용됨

Bit	7	6	5	4	3	2	1	0	
	-	-	-	-	-	-	-	RAMPZ0	RAMPZ
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- BIT 7..1 : 예약 비트
- BIT 0 : RAMPZ0(Extended RAM Page Z-pinter)

* RAMPZ0 = 0 : ELPM/SPM 명령어에 의해서 접근 가능한 프로그램 영역은 하위 64k바이트(\$0000-\$7FFF)

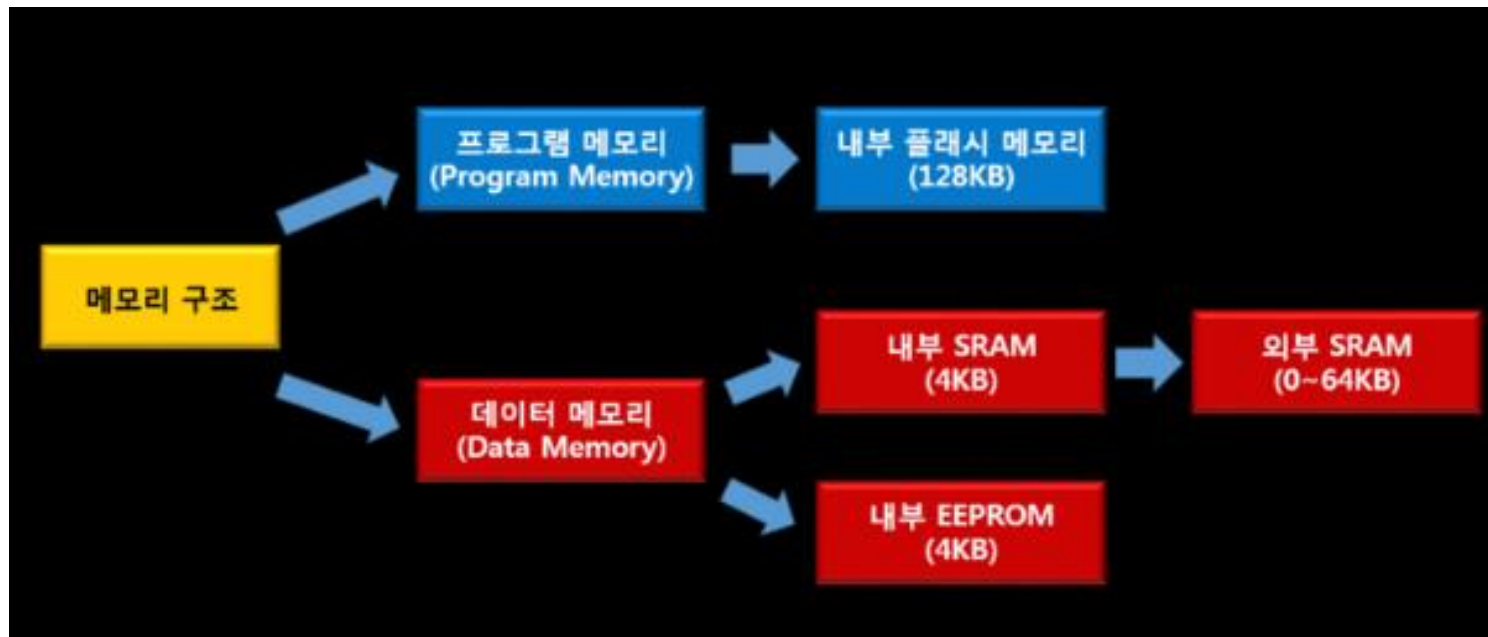
* RAMPZ0 = 1 : ELPM/SPM 명령어에 의해서 접근 가능한 프로그램 영역은 상위 64k바이트(\$8000-\$FFFF)

우선 정리만 해두어 이렇게 있다고만 알고있고 추후 External RAM을 사용할 때 참고하기!

3. Memory

1. 메모리 전체 구조

- 기능적 – 프로그램 메모리, 데이터 메모리
- 물리적 – 내부 FLASH, 내부 SRAM, 내부 EEPROM, 외부SRAM
 - FLASH : user가 작성한 프로그램 적재됨
 - 내부SRAM : 프로그램 동작에 필요한 여러정보가 동적으로 생성
 - 외부SRAM : 내부 SRAM만으로 부족할 경우
 - 내부EEPROM : 영구적으로 저장할 필요가 있는 데이터 저장



Q) 여기서 Program Memory와 Data Memory가 따로 있어서 하버드 구조인것인가?

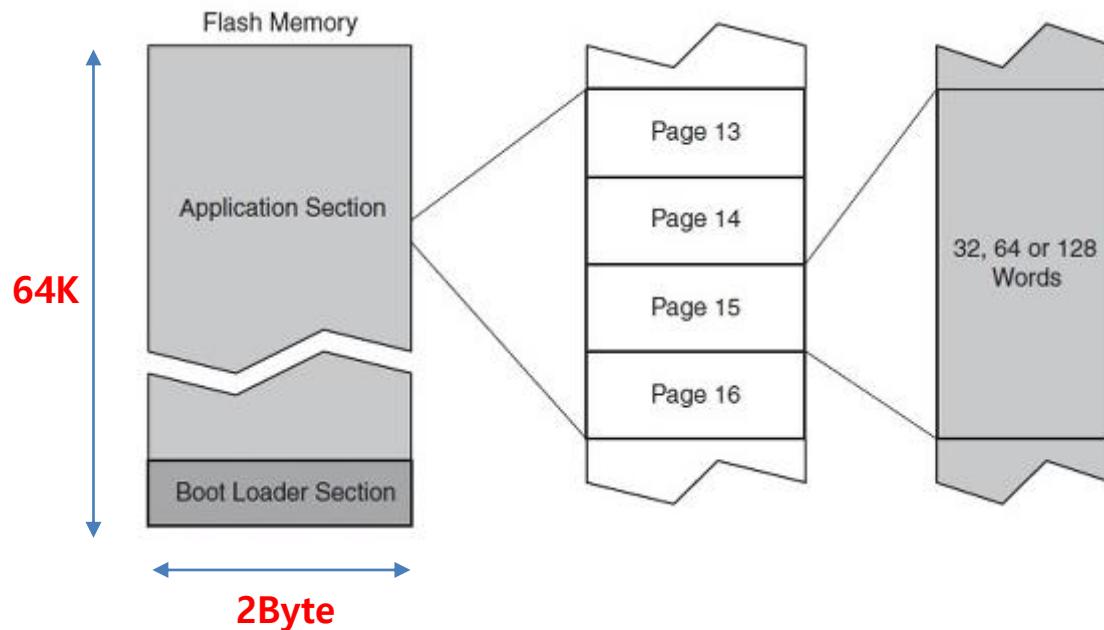
A) 메모리가 하나여도 Instruction Address/Data 와 Data Address/Data가 분리되어 있으면 하버드 구조 인데 AVR에서는 메모리가 분리되어 있어 각각의 Address/Data가 존재하고 하나는 Instruction을 하나는 Data를 가져오기 때문에 하버드 구조라고 볼 수 있음

Q) SRAM에 text(Code 영역)복사가 선택이긴 하지만 SRAM에 복사한 경우속도가 더 빠르는데 복사가 되면 SRAM과 Core간은 하버드 구조라 말할 수 있는것인가?(AVR은 복사자체가 안되나?)

3. Memory

2. Flash Program Memory

- 128K x 1Byte = 128KB (x)
- 64K x 2Byte = 128KB (o)



- 보통의 SRAM메모리는 가로 1Byte 구조로 표현
AVR은 명령어가 16비트여서 가로 2Byte 구조로 표현함

Q)뒤에 설명할 Data Memory도 가로 1Byte 구조인데 원래 Flash Program Memory는 명령어 사이즈에 따라 가로 크기 표현이 다른지 다른 프로세서 확인해볼까?

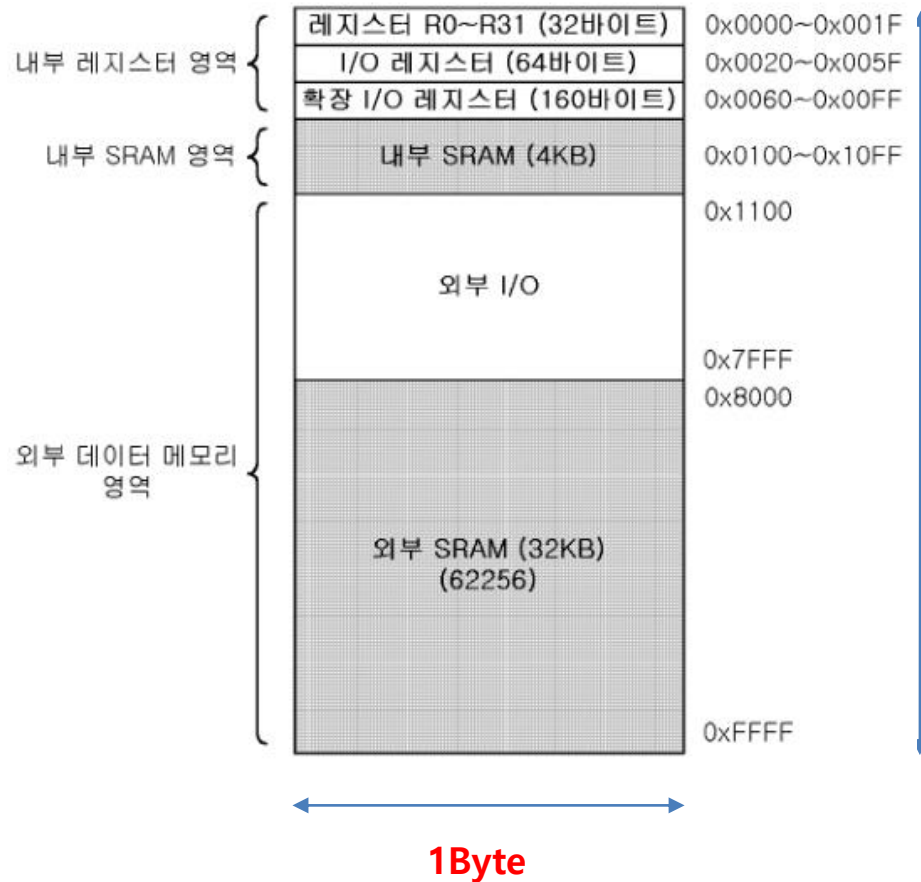
=>

ATmega128	Address bus	Data bus
프로그램 메모리	16 bit	16 bit
데이터 메모리	16 bit	8 bit

- Boot Flash Section이 보통 맨 앞에 있는 경우가 많은데 AVR은 Application Flash Section이 앞에 있음(중요x)
- 주소로 접근이 가능한 것으로 볼때 Nor Flash Memory 구조로 메모리 맵에 포함가능
- 모두 Erase 한번에 하고 새로운 데이터로 업데이트 주로 하는 편
- 기존 데이터 일부만 변경하고 싶을 경우Page단위로 Erase시키고 page buffer에 새로운 데이터 쓰고 옮기면 새로운 데이터 업데이트 가능

3. Memory

3. SRAM Data Memory



- 레지스터 **R0 - R31** : 뒤에 설명
- I/O 레지스터 : I/O장치를 제어하기 위한 레지스터
 - *I/O영역은 **Non-cacheable**로 설정해야 함
 - *I/O영역 변수는 **volatile**로 선언해야함
 - => **cache가 작동하면 메모리에 반영된 HW 변경사항을 check 못할 수도 있음**
- 내부 SRAM : Data Segment(bss, data, heap, stack...)
 - Q) text영역은 SRAM에 복사되는가?**
 - A) Atmega에서는 뒤에서 조사한 XIP 방식으로 Flash에서 명령어를 불러와 동작시킴**

참고. EEPROM(Atmega128 경우)

4KB 따로 내장되어 있음.

데이터 메모리 어드레스 영역과 별개의 영역에 위치

1Byte씩 Access 가능

3. Memory

3. SRAM Data Memory

b7	b6	b5	b4	b3	b2	b1	b0	addr.	
									R0
								\$00	
									R1
								\$01	
									⋮
									R14
								\$0E	
									R15
								\$0F	
									R16
								\$10	
									R17
								\$11	
									⋮
									R24
								\$18	
									R25
								\$19	
									R26 (XL)
								\$1A	X레지스터 하위바이트
									R27 (XH)
								\$1B	X레지스터 상위바이트
									R28 (YL)
								\$1C	Y레지스터 하위바이트
									R29 (YH)
								\$1D	Y레지스터 상위바이트
									R30 (ZL)
								\$1E	Z레지스터 하위바이트
									R31 (ZH)
								\$1F	Z레지스터 상위바이트

- **범용 레지스터**

ALU가 산술/논리 연산을 할 때 피연산자를 여기서 일경오며 연산결과도 여기로 저장

예) $R0 = R0 + R1$ ($R0, R1$ 에는 add 연산의 피연산자들)

- **어셈블러에서 아래와 같이 참조**

0x00 – 0x1F 같은 주소로 참조(x)

R0 – R31 이름으로 참조(o)

- **상위 R0 – R15는 직접 상수 대입 불가
하위 R16 – R31에만 직접 상수 대입 가능**

- R24 – R31은 2개씩 묶어서 16bit 단위로 연산을 하기 용이하며 각가 **X, Y, Z**레지스터로 명명하여 **16bit 연산** 뿐만 아니라 주소를 지정하거나 **주소 연산**을 하는데도 사용

참고. XIP vs RAM Loading

1. 임베디드 시스템 응용 프로그램 구동 방식

XIP(eXecute In Place)

- 대개 비휘발성 코드 메모리(ROM, NOR Flash)에 프로그램이 기록됨
- 실행시 그냥 그자리(Reset Vector)에서 프로그램이 실행됨
- 대부분 기존 임베디드 시스템에 적용되던 방식(일반적인 마이컴)

즉, **text 영역에 들어가는 code, Ro data가 그대로 ROM에 유지**

RAM Loading

- 응용 프로그램이 어디에 있었던지 실행전에는 RAM으로 옮겨짐
- 실행시 RAM에 있는 상태로 실행
- 일반적으로 Non-Bootable 메모리에 실행파일이 저장되어 있는 경우 사용하는 모드
- Non-Bootable 메모리: **NAND Flash**, HDD, SSD, SD Card, MMC Card, eMMC 등
- Bootable

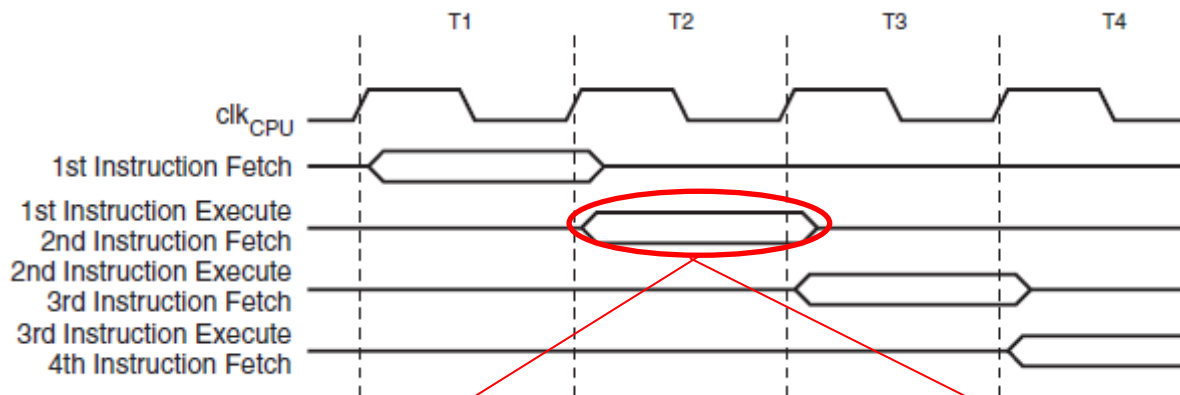
참고. 1MIPS/1MHz(AVR)

1. 병렬 명령 fetch 및 execution

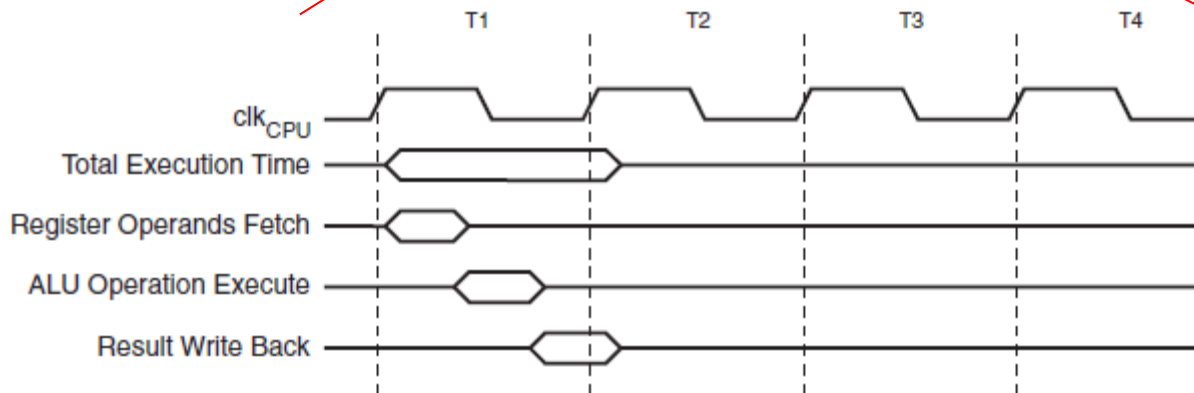
1MIPS = 1 Million Instruction Per Second의 축약어로 1초에 100만개 명령어 처리

1MHz = 1초에 100번 clock

즉, 1MIPS/1MHz 는 1 clock에 1 Instruction 처리한다는 의미



병렬 명령 패치 및 실행



Q) 1clock에 1Instruction 처리를 어떻게 하나?

A) 1개의 명령처리는 4clock을 사용하지만 위의 그림보면 4개까지 병렬 처리가 가능하여 4번째 명령 처리부터는 1 clock에 1개의 Instruction 처리하는 효과가 있게 됨 (즉, 파이프라인 도움으로 가능함)

Q) 여기서 Fetch는 IF/Decoding, Execute는 Operand/ALU/WriteBack를 포함하여 말하는 것인가?

A)

4. I/O Ports

1. I/O port 구조

Port A Data Register –
PORTA

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Port A Data Direction
Register – DDRA

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Port A Input Pins
Address – PINA

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

DDR(Data Direct Register) :

해당 Port의 각 Pin을 입/출력 설정 하는데 사용 하는 Rg.

PORT(Data Register) :

Microprocessor 내부 Rg로 부터 Data를 받아 외부 Interface 회로에 Data를 출력 하는 Rg

PIN(Port Input Pins Register) : 입력 핀으로 부터 Data를 받아 Input 명령이 실행 될 때 Microprocessor 내부 Rg에 Data를 전달 하는 Rg

- Port A 부터 Port G(Port G Rg는 5 Bit 구조 임) 까지 7개 Rg 가 모두 같은 구성과 기능
- PORTA와 DDRA Rg는 Rg에 저장된 내용을 읽거나(Read) 새로운 Data를 쓸(Write) 수 있음
- **PINA Rg는 읽기(Read)만 가능 하고, Data를 쓸(Write) 수 없음**
- Reset 시 PORTA와 DDRA Rg는 0로 초기화, 그러나 PINA Rg는 초기화 x

4. I/O Ports

2. I/O Port 설정

1) Output Port로 사용하기 위한 설정과 동작

Output Port로 사용 하고자 하는 Port의 DDR Rg에 1을 Write 하여 해당 Port를 Output Port로 설정 한다.
Output Port에 Data를 출력 한다.

PF0(Port F의 0번 Bit)를 Output Port로 설정 하고 PF0에 1을 출력 하는 Coding 예

```
DDRF |= 0x01; // PF0를 Output Port로 설정 한다. DDRF의 0번째 Bit가 1로 설정 됨.
```

```
PORTF |= 0x01; // PF0 값이 1이 된다. PF0 이외 다른 Bits는 변동 되지 않는다.
```

2) Input Port로 사용하기 위한 설정과 동작

Input Port로 사용 하고자 하는 Port의 DDR Rg에 0을 Write 하여 해당 Port를 Input Port로 설정 한다.
Input Port(PIN)으로 부터 Data를 입력 받는다.

PD0(Port D의 0번 Bit)를 Input Port로 설정 하고 PIND의 0번 Bit의 Data를 입력 받아 PF0에 출력하는 Coding 예

```
DDRD &= (~0x01); // PD0를 Input Port로 설정 한다.
```

```
DDRF |= 0x01; // PF0를 Output Port로 설정 한다.
```

```
PORTF &= (~0x01); // Port F의 PF0를 0로 Clear 한다.
```

```
PORTF |= (PIND & 0x01); // PF0 이외 PORTF 다른 Bits 값은 변경되지 않고, PF0 Bit 만 PIND의 0번 Bit(PD0) 값과 같도록 Setting 된다.
```

3) Pull-up Resistor 사용을 위한 설정

Pull-up Resistor는 Switch 등을 연결하는 경우 외부에서 부가되는 회로를 최소화 하기 위한 것 이다. 아래 Switch Interfacing Circuit 예를 참고 할 것.
윗 회로에서 Pull-up Resistor를 Active 상태로 하는 FET Switch를 Turn On 상태로 하기 위한 조건은 PUD이 0이고, DDRx Rg의 출력이 0, PORTx의 출력이 0 인 상태 이다.

PD0(Port D의 0번째 Bit)에 Push Button Switch를 연결 하고 이 SW의 상태를 읽어 PF0에 출력 하는 Coding 예

```
DDRD &= (~0x01); // PD0를 Input Port로 설정 한다.
```

```
PORTD |= 0x01; // Pull-up 저항을 Active 상태로 하기 위함. PUD는 Reset 시 0으로 초기화 되기 때문에 이 예에서는 PUD를 0로 하는 Coding을  
생략 하였다.
```

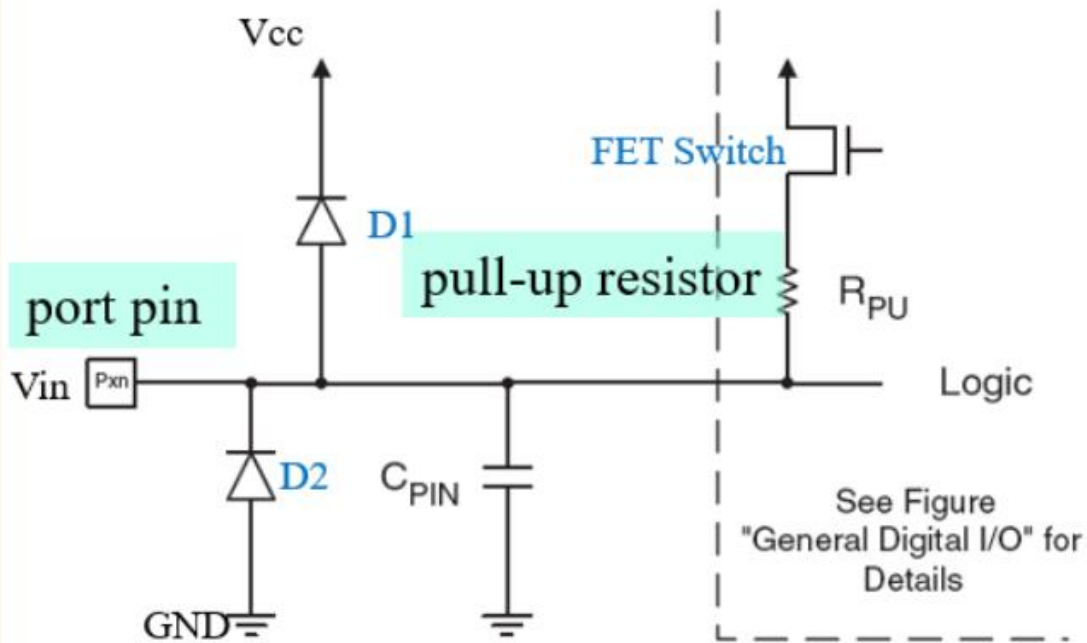
```
DDRF |= 0x01; // PF0를 Output Port로 설정 한다. DDRF의 0번째 Bit가 1로 설정 됨.
```

```
PORTF &= (~0x01); // Port F의 PF0를 0로 Clear 한다.
```

```
PORTF |= (PIND & 0x01); // PIND의 0번 Bit 값과 PF0 값이 같도록 Setting 한다.
```

4. I/O Ports

3. I/O Port 보호 및 Pull-up Resistor



Protection Diodes

Pin에 인가 되는 전압이 과도하게 높아 지거나 낮아 지지 않도록 하여 I/O Port회로를 보호

Programmable Pull-up resistor

FET Switch 'On','Off'를 통하여 pull-up resistor 설정하며 외부 pin이 open 상태이면 pin 전압은 High가 됨

4. I/O Ports

4. SFIOR: Special Function IO Reg

Bit	7	6	5	4	3	2	1	0	
	TSM	–	–	–	ACME	PUD	PSR0	PSR321	SFIOR
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Pull-up disable Bit(PUD): I/O Port의 Pull-up 기능을 제어(Enable or Disable 상태로 설정) 하는 Bit

Bit 0 : DDRx가 0, PORTx가 1 이면 I/O Port의 **Pull-up 기능을 Enable** 상태

Bit 1 : DDRx와 PORTx 의 상태에 상관 없이 I/O Port의 **Pull-up 기능은 Disable** 상태

*SFIOR의 다른 제어 Bit는 Timer(TSM, PSR0, PSR123)와 A/D 변환기(ACME)의 제어에 사용됨

Table 25. Port Pin Configurations

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

QuickSort.c

```
/*
 * Quick Sort 알고리즘 구현 순서 고민(그림 그려가며 해결)
 * 1) Pivot 기준 위치 정하기
 * 2) left, right, pivot 변수 및 위치 설정
 * 3) SWAP: left <-> right)
 * 4) SWAP: pivot <-> i=j(right)
 * 5) pivot 기준으로 좌측, 우측 호출하며 재 반복(즉, 재귀함수 사용)
 * KEY : sort 함수, swat함수, 재귀호출
 */
```

```
#include <stdio.h>
```

```
void quick_sort(int *pArr, int start, int end);
```

```
void SWAP(int *, int *);
```

```
void printf_Arr(int *pArr, int num)
```

```
{
    int i;

    for(i = 0; i < num; i++)
    {
        printf("%d ", pArr[i]);
    }
    printf("\n");
}
```

```
70 30 60 50 80 20 10 20
10 20 20 30 50 60 70 80
```

```
int main(void)
```

```
{
    int arr[] = {70, 30, 60, 50, 80, 20, 10, 20};
    int start = 0; //시작위치(left)
    int num = sizeof(arr)/sizeof(arr[0]);
    int end = num - 1; //끝 위치(right)

    printf_Arr(arr, num);
    quick_sort(arr, start, end);
    printf_Arr(arr, num);

    return 0;
```

```
void quick_sort(int *pArr, int start, int end)
{
```

```
    int left = start;
```

```
/*
```

```
 * right = end - 1    ==>    right = end
```

```
 * right--가 포함된 while문을 돌게 함으로써 60 60 남았을 때 60이 pivot ==left == right가 되어
```

```
 * 2번 swap해도 변화가 없게된다.
```

```
 * 단점은 불필요한 Swap을 계속적으로 해주기 때문에 이러한 경우에는 부득이하게 조건문을 만족할 때 swap하는 식으로
```

```
 * 변경 해야 할 것 같음
```

```
*/
```

```
int right = end; //right--가 포함된 while문을 돌게 함으로써 left = right = pivot 이 되어 Swap 하더라도
```

```
int pivot = end; //배열 끝값을 Pivot 기준으로 설정
```

```
if(left >= right) return;
```

```
while(left < right)
```

```
{
```

```
/*
```

```
 * while 대신 if 넣으면 호출이 떨어지는 이유는?
```

```
 * >>> 29번째 while문 돌고 if 체크하고 2번 조건을 반복하며 left를 증가시킴
```

```
 * >>> while문으로 하면 해당 구문 1번 조건만 돌면서 left를 증가시킴
```

```
*/
```

```
while(pArr[left] < pArr[pivot] && left < right)
```

```
{
```

```
    left++;
```

```
}
```

```
while(pArr[right] >= pArr[pivot] && left < right)
```

```
{
```

```
    right--;
```

```
}
```

```
SWAP(&pArr[left], &pArr[right]);
```

```
}
```

```
SWAP(&pArr[left], &pArr[pivot]);
```

```
//printf_Arr(pArr, 8);
```

```
quick_sort(pArr, start, left-1);
```

```
quick_sort(pArr, left+1, end);
```

```
}
```

```
void SWAP(int *A, int *B)
```

```
{
```

```
    int temp;
```

```
    temp = *A;
```

```
    *A = *B;
```

```
    *B = temp;
```

아두이노-PIR_SENSOR_OFF

모션인식 OFF 상태/LED 'OFF'/SERIAL '0'

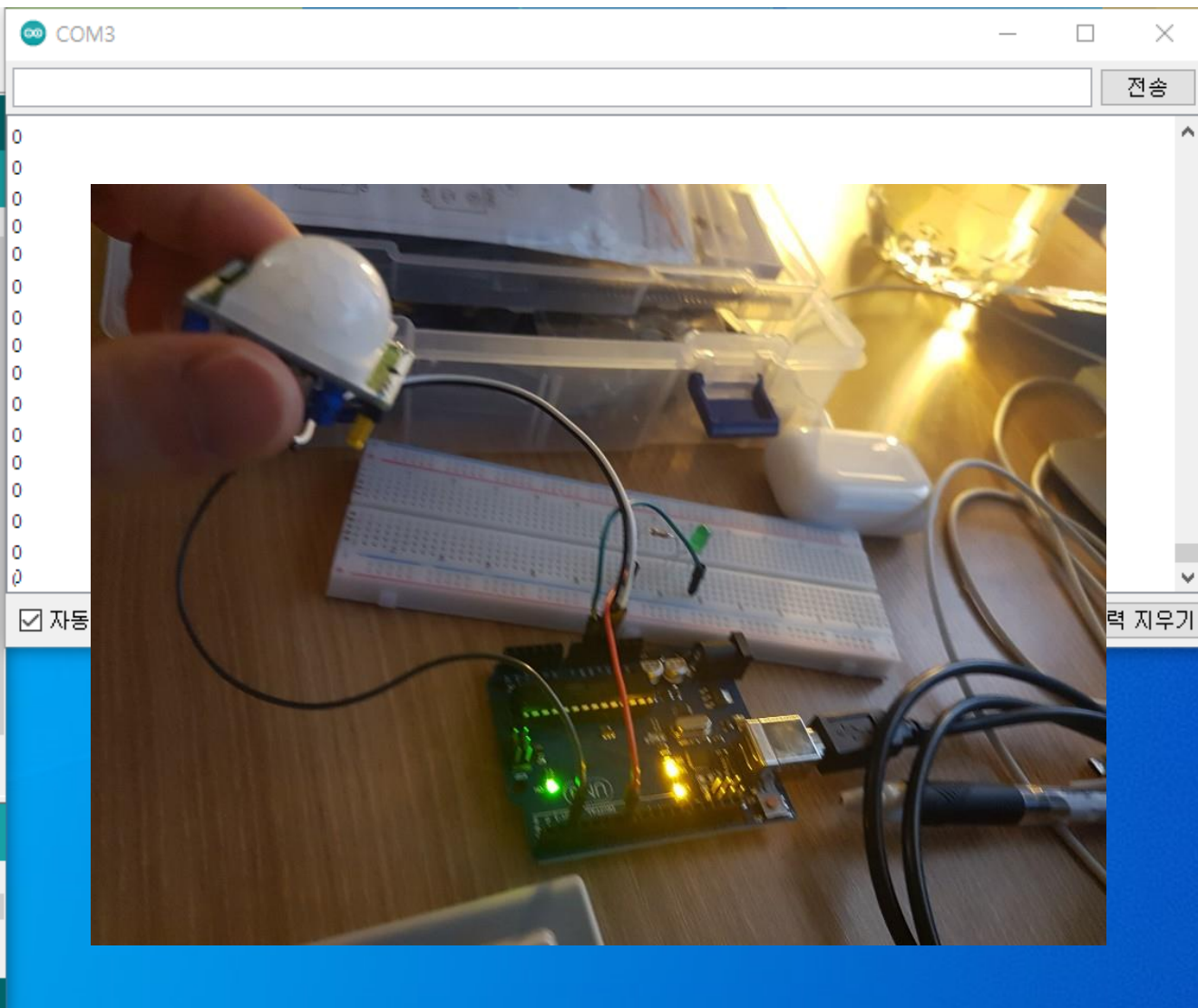
```
02_PIR_SENSOR_ON_OFF | 아두이노 1.8.13
파일 편집 스케치 툴 도움말

02_PIR_SENSOR_ON_OFF $
int val = 0;
void setup() {
  pinMode(2, INPUT); //디지털 2번핀은 입력모드
  pinMode(7, OUTPUT); //디지털 3번핀은 출력모드
  Serial.begin(9600); //시리얼 통신 시작, 통신속도 9600
}

void loop() {
  val = digitalRead(2);
  if(val == HIGH)
  {
    digitalWrite(7, HIGH);
  }
  else
  {
    digitalWrite(7, LOW);
  }
  Serial.println(val);
}

업로드 완료.
스케치는 프로그램 저장 공간 2004 바이트 (6%)를 사용. 최대 32256 바이트.
전역 변수는 동적 메모리 190바이트 (9%)를 사용, 1858바이트의 지역변수가 남음

12 Arduino Uno on COM3
```



아두이노-PIR_SENSOR_ON

모션인식 ON 상태/LED 'ON'/SERIAL '1'

02_PIR_SENSOR_ON_OFF | 아두이노 1.8.13

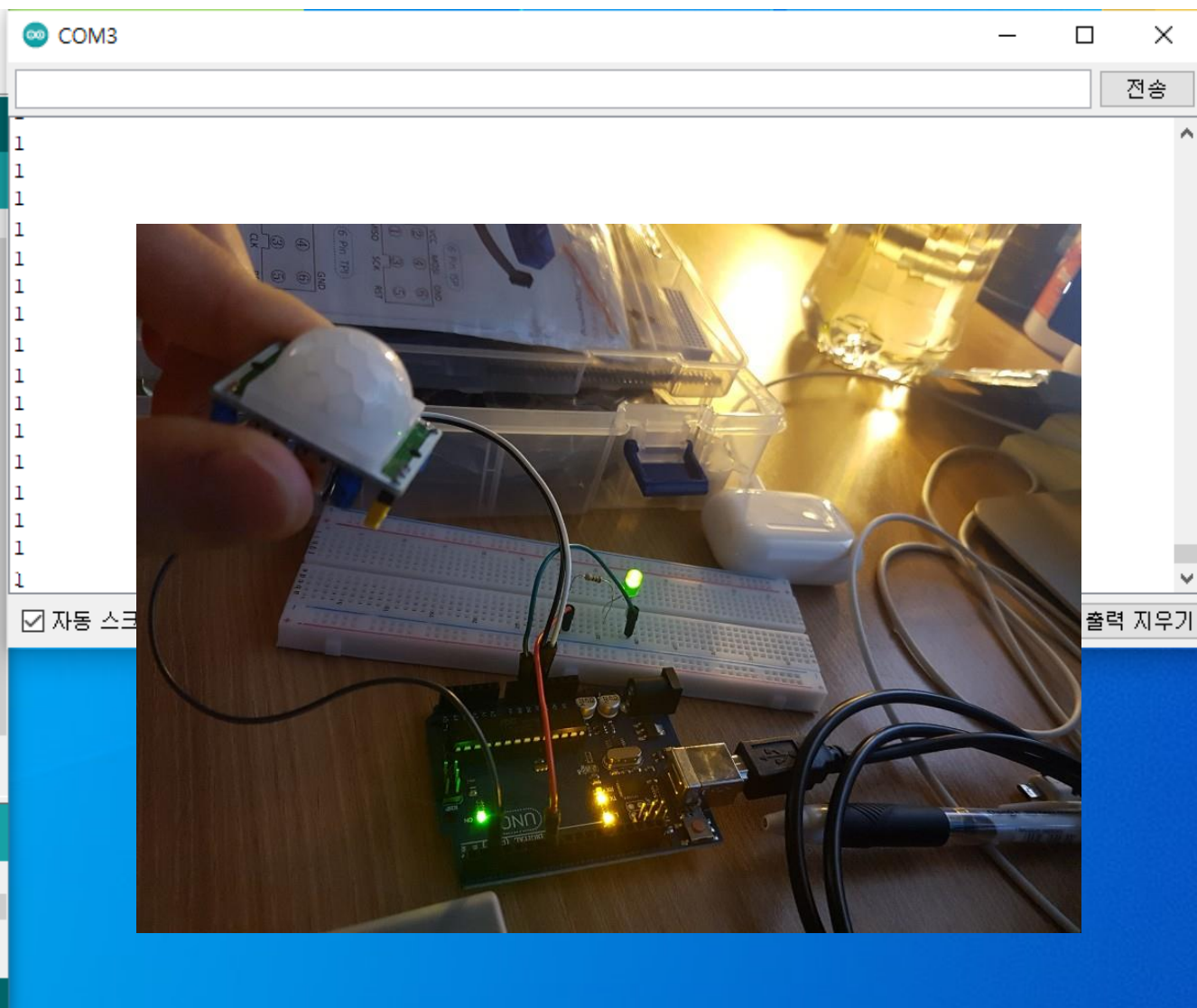
파일 편집 스케치 툴 도움말

```
02_PIR_SENSOR_ON_OFF $  
  
int val = 0;  
void setup() {  
  pinMode(2, INPUT); //디지털 2번핀은 입력모드  
  pinMode(7, OUTPUT); //디지털 3번핀은 출력모드  
  Serial.begin(9600); //시리얼 통신 시작, 통신속도 9600  
}  
  
void loop() {  
  val = digitalRead(2);  
  if(val == HIGH)  
  {  
    digitalWrite(7, HIGH);  
  }  
  else  
  {  
    digitalWrite(7, LOW);  
  }  
  Serial.println(val);  
}
```

업로드 완료.

스케치는 프로그램 저장 공간 2004 바이트 (6%)를 사용. 최대 32256 바이트.
전역 변수는 동적 메모리 190바이트 (9%)를 사용, 1858바이트의 지역변수가 남음

12 Arduino Uno on COM3



아두이노-HC_distance_sensor_with_PIR

Distance >= 20 / LED 'OFF'

```
03_HC_distance_sensor_with_PIR | 아두이노 1.8.13
파일 편집 스케치 툴 도움말

03_HC_distance_sensor_with_PIR
int trig = 6;
int echo = 5;

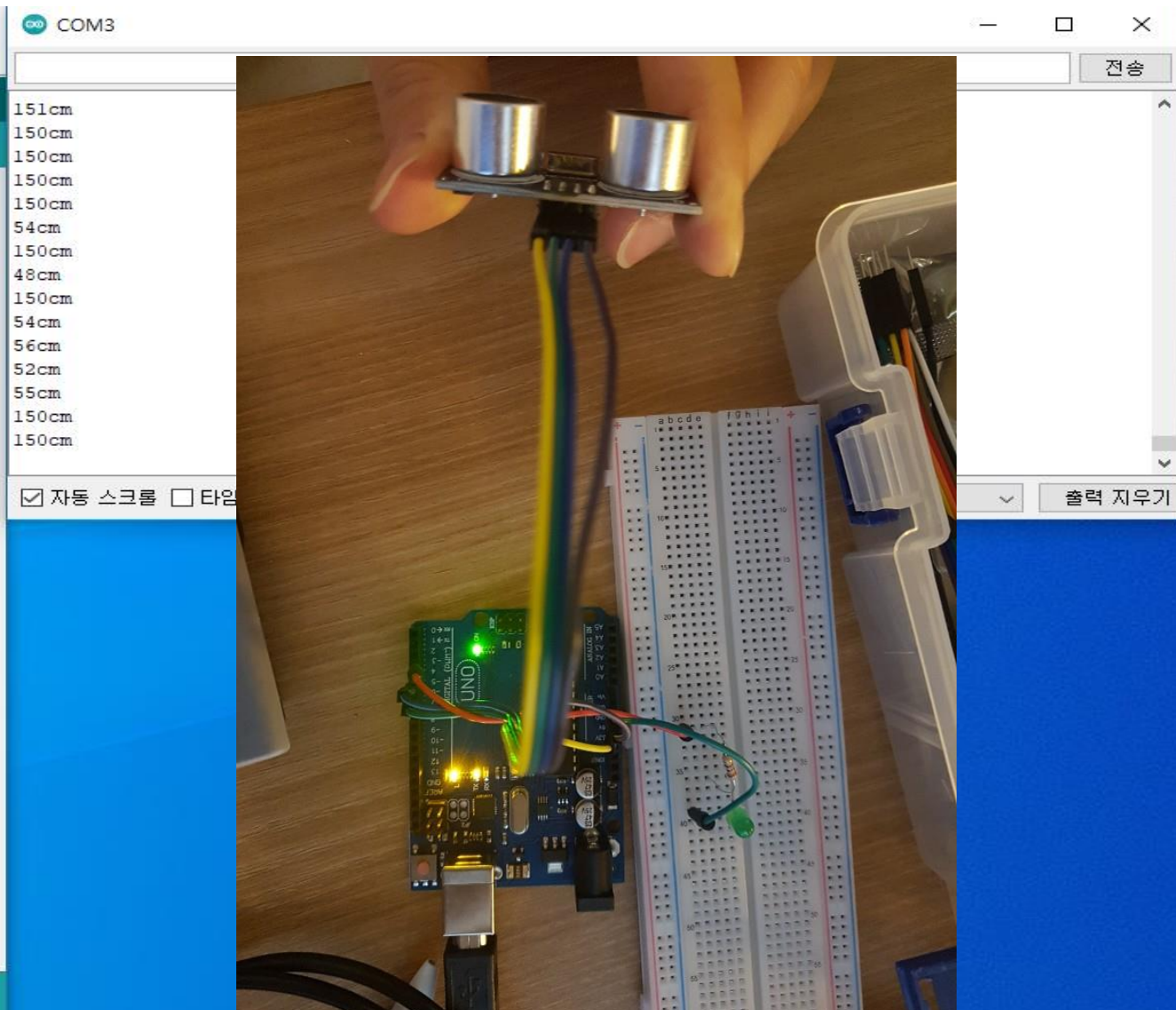
void setup() {
  pinMode(3, OUTPUT);
  Serial.begin(9600);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
}

void loop() {
  digitalWrite(trig, HIGH);
  delayMicroseconds(10); //10us
  digitalWrite(trig, LOW);

  //20Mhz = 340m/s, m -> cm (x10000), around trip -> /2
  int distance = (pulseIn(echo, HIGH)/1000000)*(340 *100)/2; //pus

  Serial.print(distance);
  Serial.println("cm");
  delay(100);

  if(distance < 20)
  {
    digitalWrite(3, HIGH);
  }
  else
  {
    digitalWrite(3, LOW);
  }
}
```



아두이노-HC_distance_sensor_with_PIR

Distance<20/LED 'ON'

03_HC_distance_sensor_with_PIR | 아두이노 1.8.13

파일 편집 스케치 툴 도움말

03_HC_distance_sensor_with_PIR

```
int trig = 6;
int echo = 5;

void setup() {
  pinMode(3, OUTPUT);
  Serial.begin(9600);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
}

void loop() {
  digitalWrite(trig, HIGH);
  delayMicroseconds(10); //10us
  digitalWrite(trig, LOW);

  //20Mhz = 340m/s, m -> cm (x10000), around trip -> /2
  int distance = (pulseIn(echo, HIGH)/1000000)*(340 *100)/2; //pus

  Serial.print(distance);
  Serial.println("cm");
  delay(100);

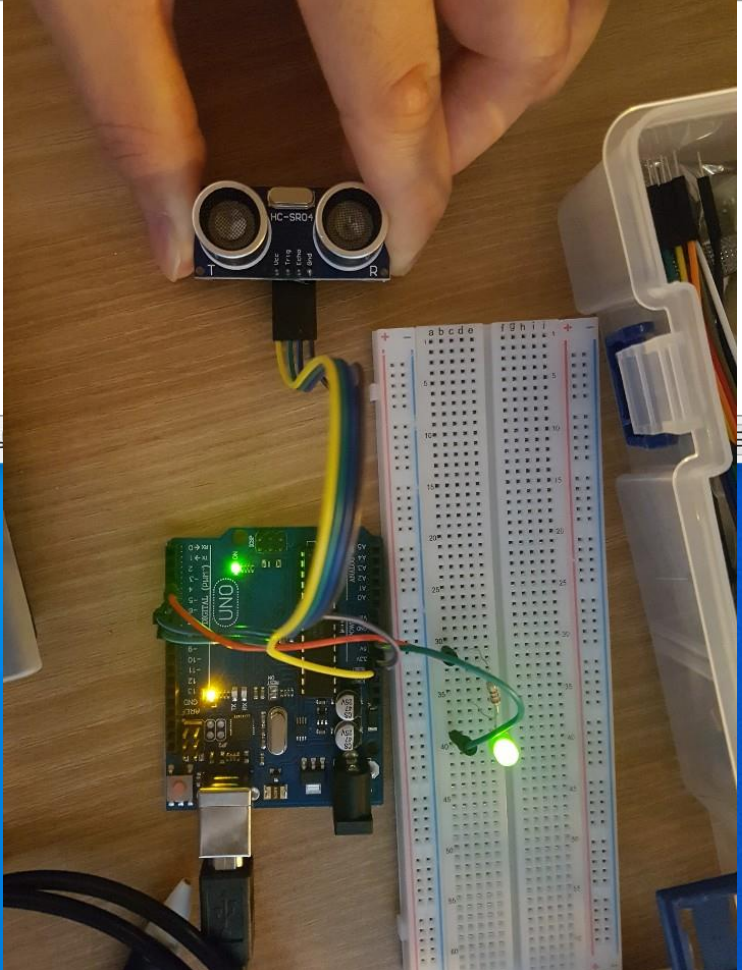
  if(distance < 20)
  {
    digitalWrite(3, HIGH);
  }
  else
  {
    digitalWrite(3, LOW);
  }
}
```

COM3

4 p

2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm
2cm

☒ 자동 스크롤 ☐ 타임



트레이트 출력 지우기



감사합니다.