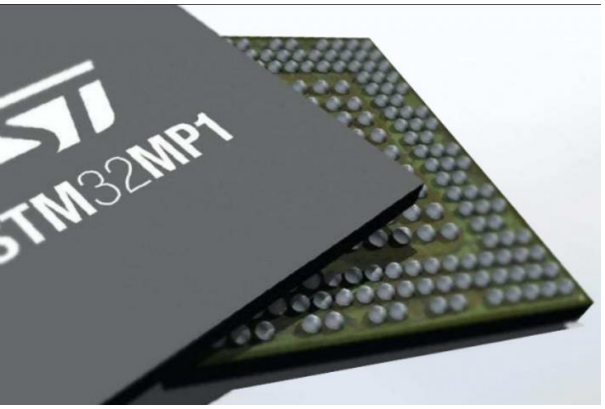
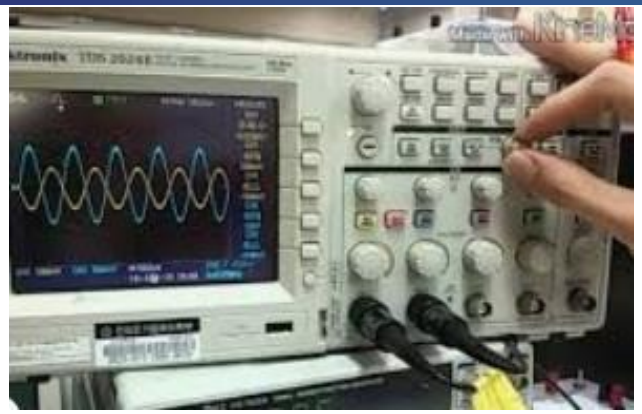
호서대학교
HOSEO UNIVERSITY





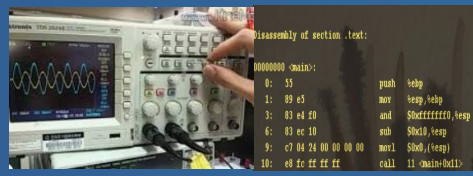
```
Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 e4 f0                and    $0xfffffff0,%esp
   6:   83 ec 10                sub    $0x10,%esp
   9:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  10:   e8 fc ff ff ff          call   11 <main+0x11>
```

전력 부채널 신호를 이용한 명령어 역어셈블러

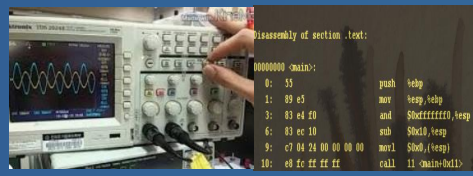
2022. 7.

호서대학교 하재철

순 서



I. 하드웨어 역어셈블러 개요



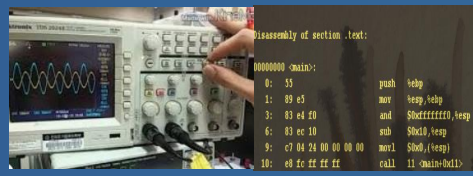
역어셈블러(Disassembler)?

- 기계어를 어셈블리어로 변환하는 프로그램
  - 0100101011…… -> MOV r18, r20;

부채널 정보 기반 명령어 역어셈블러?

- 역어셈블링 대상이 기계어가 아닌 부채널 누출 정보
- MCU의 소비 전력, EM 신호 등으로 부터 디바이스 내부에서 구동되는 명령어 복구
- 머신러닝을 이용한 프로파일링 부채널 공격과 유사
  - 명령어에 대한 템플릿 생성 후 명령어 복구

I. 하드웨어 역어셈블러 개요

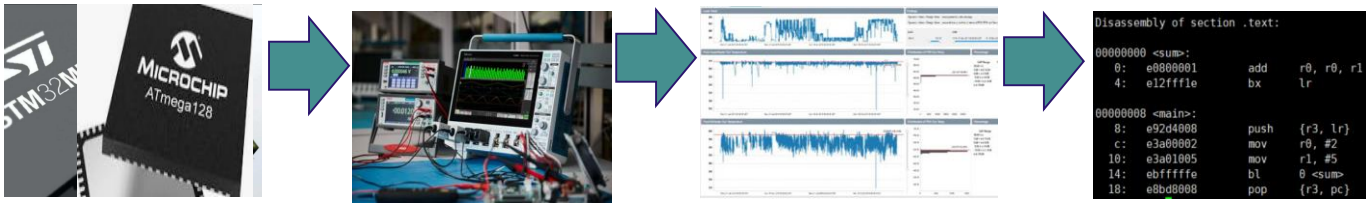


❖ 연구 개발의 중요성 및 활용

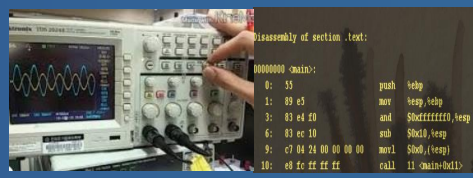
- 하드웨어 장치내 멀웨어 탐지
- 역공학에 기반하여 불법 IP 사용 탐지
- 공개되지 않은 암호 메커니즘 분석
- 하드웨어-펌웨어 공동 증명
- 소프트웨어 기반 마이크로아키텍처 공격 탐지

❖ HW에 대한 역공학 수단으로 부채널 정보를 사용

➔ 부채널 기반 역어셈블러(Side-Channel Based Disassembler)



I. 하드웨어 역어셈블러 개요



❖ 부채널 기반 역어셈블러 구현 사례

- 1) Vermoen 등(2007년)

- 자바 스마트 카드 대상, 부채널 정보 상관 계수를 이용
- 10개의 명령어 클래스를 92% 정확도로 분류

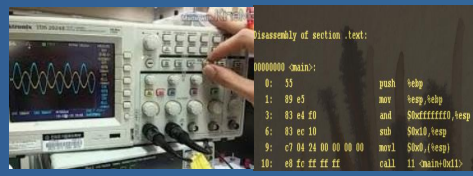
- 2) Eisenbarth 등(2010년)

- 마이크로 칩 PIC16F687 프로세서 대상, 템플릿 공격 시도
- 33개의 명령어를 70.1% 정확도로 구별

- 3) Msgna 등(2014년)

- ATMega163 프로세서 대상, k-NN 분류 알고리즘 이용
- 39개의 명령어를 100% 정확도로 분류

I. 하드웨어 역어셈블러 개요



❖ 부채널 기반 역어셈블러 구현 사례

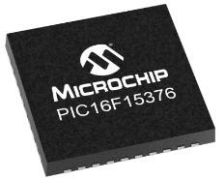
- 4) Strobel 등(2015년)

  - PIC16F687 대상 전자기파 이용, k-NN 분류 알고리즘 이용
  - 33개의 명령어를 96.2% 정확도로 구별

- 5) Park 등(2018년)

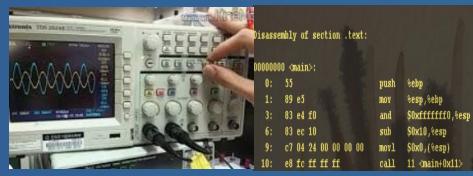
  - ATMega328 프로세서 대상, PCA 및 CWT 등 잡음 감소
  - 112개 명령어/레지스터 64개를 99.0% 정확도로 분류

- 6) Cristiani 등(2019년)



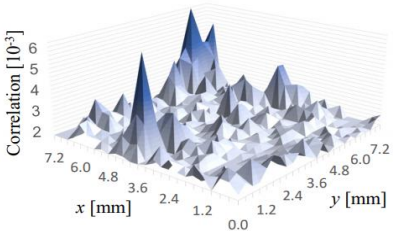
  - PIC16F15376 대상 전자기파 이용,
  - 비트 수준에서 99.4%, 14비트 명령어 50개에서 95% 정확도

I. 하드웨어 역어셈블러 개요



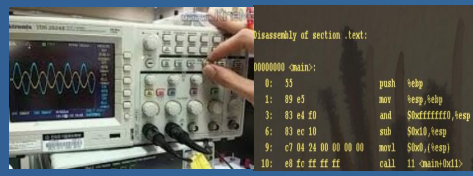
❖ 부채널 기반 역어셈블러 연구 동향

- 전력 신호 이용 vs. 전자기파 신호 이용



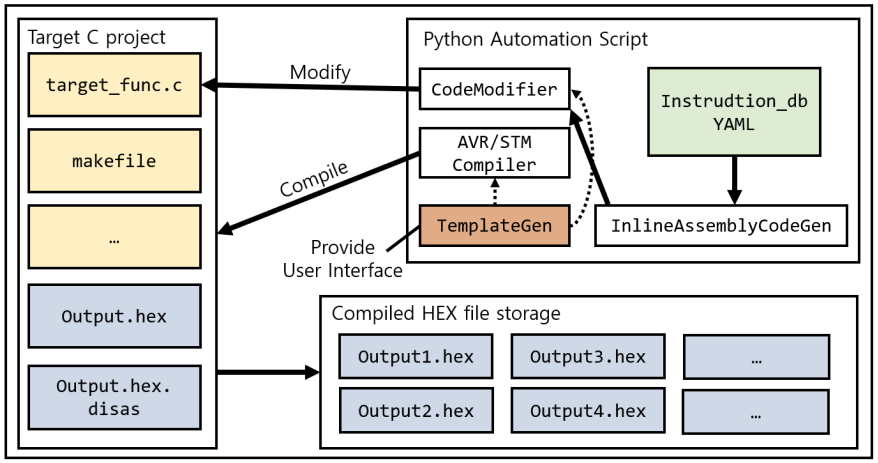
- MCU 종류 : 8비트 프로세서(ATMega, PIC…) ➔ 32비트
- 신호 차원 축소 및 잡음 감소 처리(PCA, LDA…)
- Classifier : 상관계수, 머신 러닝/딥 러닝(k-NN, MLP, MLP…)
- 비트 수준 vs. 명령어 수준 분석

II. 역어셈블러 구축 과정

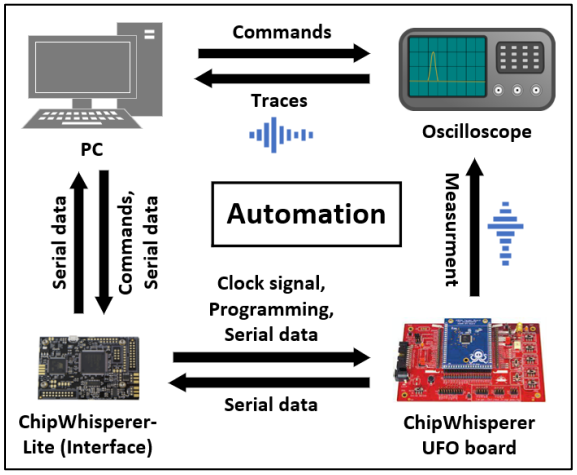


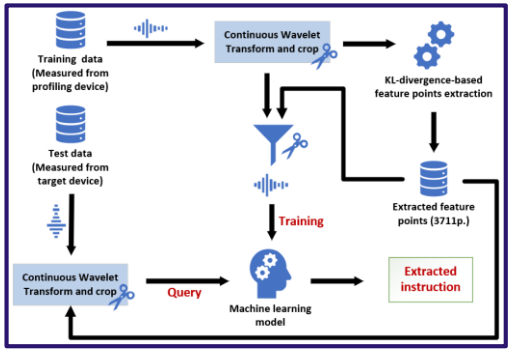
❖ 명령어 역어셈블러 구축 과정

1) 명령어 템플릿 구성을 위한 펌웨어 생성 자동화



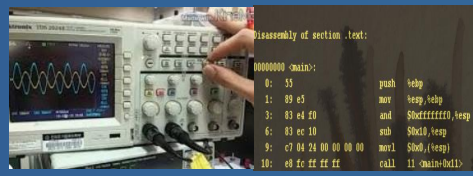
2) 명령어 파형 수집 자동화





3) 사전 신호 처리 및 머신러닝 적용의 자동화

II. 역어셈블러 구축 과정



❖ 명령어 템플릿 구성을 위한 펌웨어 생성 자동화

- 측정 대상 명령어로 구성된 펌웨어 파일 필요 (.hex)

- 명령어 템플릿 생성 방법 ➔ 1)의 방법 사용

  1) Inline assembly(C언어내 어셈블러) 기반 Opcode 및 Operand 조합

     → 일반적인 Opcode 수준 분류기에서 사용

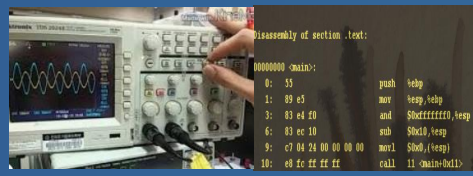
     → 분석 대상 MCU마다 문법에 맞는 어셈블리어를 직접 생성해야 하는 단점

     → 사전 분석된 XMEGA128 명령어 40개를 대상으로 펌웨어 생성

  2) 컴파일된 바이너리 파일을 직접 수정

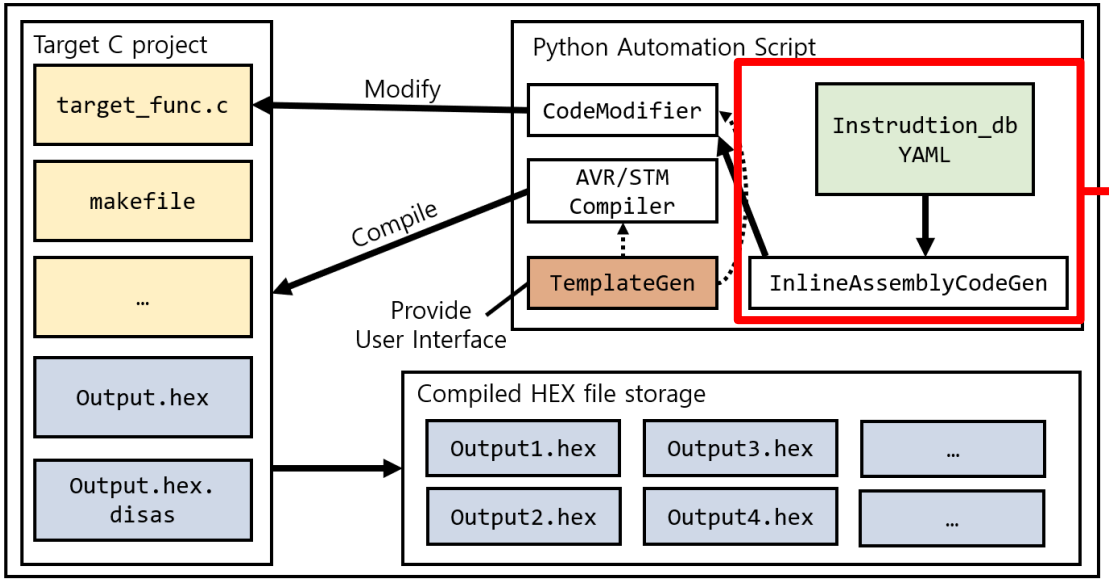
     → Opcode/operand 구분이 없는 비트 수준 분류기에서 사용

II. 역어셈블러 구축 과정



❖ 명령어 템플릿 구성을 위한 펌웨어 생성 자동화

- MCU의 주요 명령어 데이터베이스 구축(수동)
  - 명령어 데이터베이스를 기준으로 문법에 맞는 어셈블리어 생성기 제작
- 하나의 명령어 삽입 → 컴파일 → .hex 파일 저장 단계를 자동화

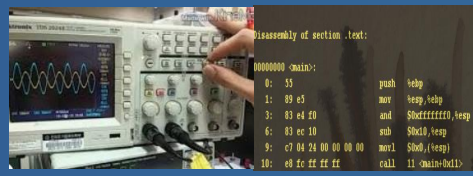


<수동으로 작성>

- 명령어 데이터 베이스 작성
- 데이터베이스 기반 어셈블리어 생성기

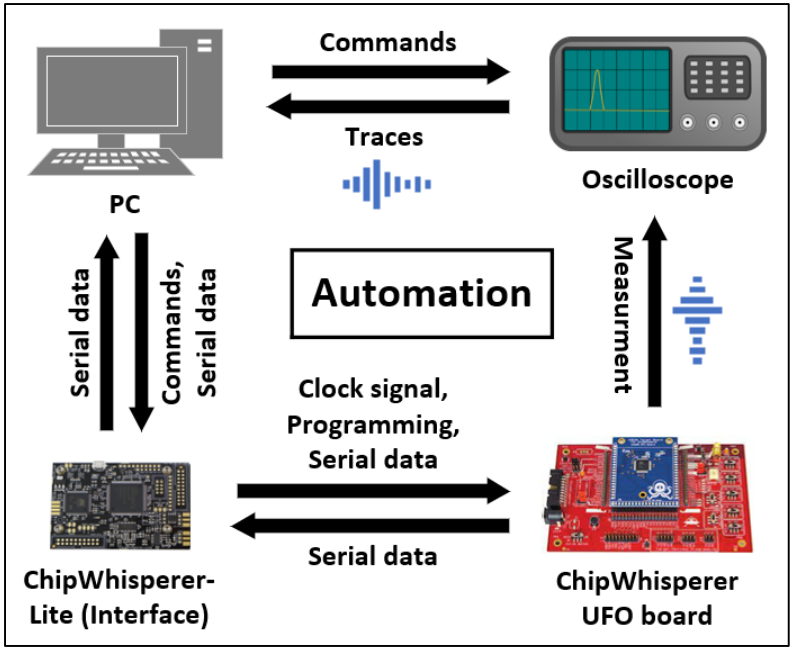
예) 40개 명령어 * 30개 시퀀스 = 1200개 hex 파일

II. 역어셈블러 구축 과정

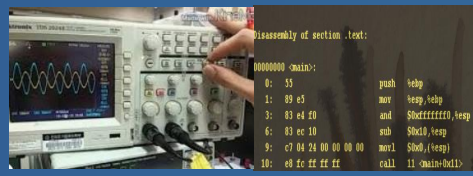


❖ 명령어 템플릿 파형 수집 자동화

- PC ↔ Profiling board
  - 템플릿 프로그램 적재
  - 시리얼 통신 / 명령어 수행
  - 동작 클럭 변경
- PC ↔ Oscilloscope
  - 명령어 통신 (Lecroy)
  - 소비 전력 파형 수신
- 파이썬 언어 기반 자동화 구축
  - 오픈 소스 활용
  - 전 과정을 자동화로 구현

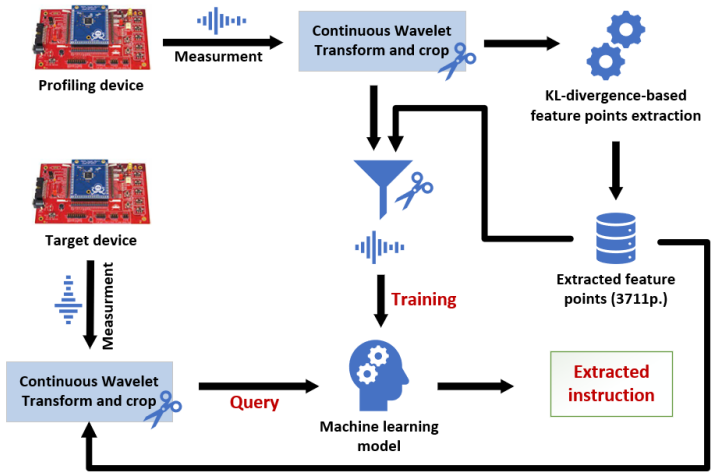


II. 역어셈블러 구축 과정

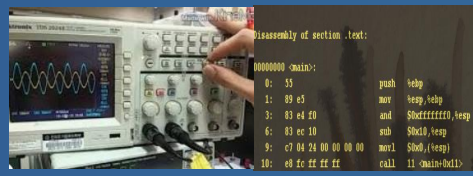


❖ 사전 신호 처리 및 머신 러닝을 적용한 분류기

1. 측정된 Raw 데이터
2. CWT 변환 및 POI 추출
   - 최적의 스케일 계산 및 CWT 변환
   - 명령어 클럭에 따른 POI 자동 추출
3. KL-Divergence 기반 특징 추출
   - KLD 분석을 통한 명령어 특징 추출
   - 파라미터 조절을 통한 특징 포인트 개수 조절 가능
4. 기계학습 및 딥러닝 모델 학습과 검증
   - 주어진 파라미터를 이용해 학습 및 검증
   - 단일 파형으로 질의하면 복구된 명령어로 응답 가능

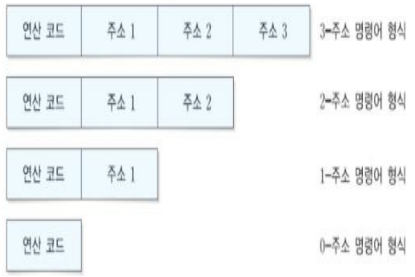


III. 명령어 분석 및 실험 환경

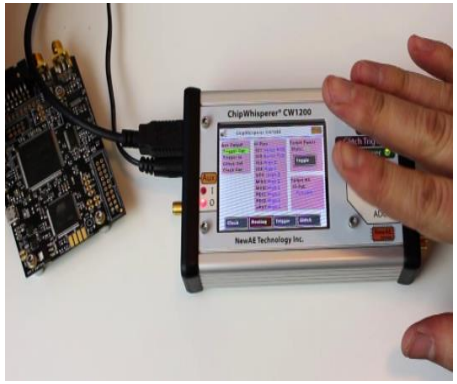


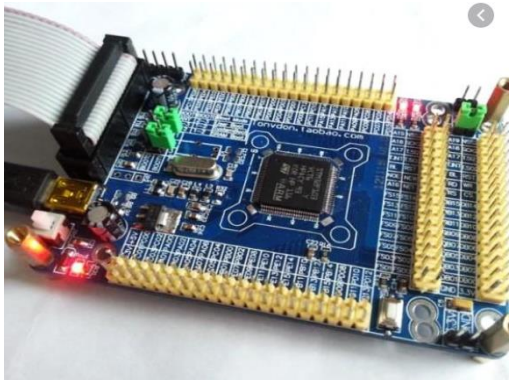
❖ 명령어 분석 및 실험 환경 구축

- 1) 대상 프로세서의 실행 명령어 코드 종류
- 2) 명령어에 대한 동작 과정 분석(파이프라인)
- 3) 부채널 분석을 위한 실험 환경 세팅 및 데이터 셋 수집

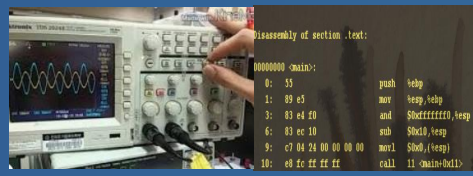


명령어 분류	어셈블리 언어로 명령어 표현
1-주소 명령어	LOAD X
2-주소 명령어	MOV X, Y
3-주소 명령어	ADD X, Y, Z



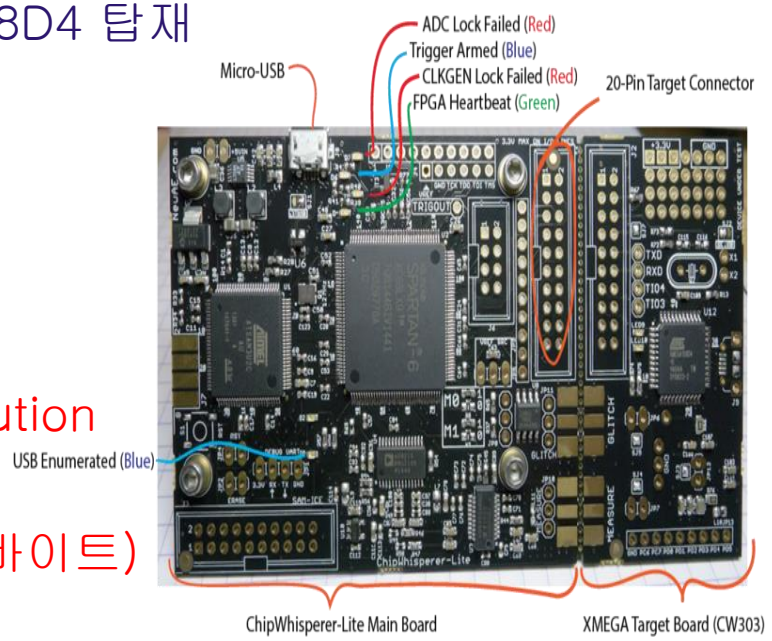


III. 명령어 분석 및 실험 환경



명령어 분석 대상 프로세서

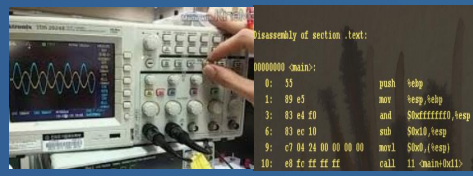
- NewAE사의 ChipWhisperer 플랫폼 사용
- 누설 전력 신호를 이용하여 분석
- CW303 타겟 보드에 Atmel XMEGA128D4 탑재



Atmel XMEGA128D4 MCU

- 8비트 단위 처리 프로세서
- Single-level 파이프라인
  - 2개의 스테이지: Pre-fetch, Execution
- 137개의 명령어 집합
  - 2바이트 기계어로 인코딩 (일부 4바이트)
- 32개의 범용 레지스터 사용
- 최대 7.37MHz 외부 클럭으로 동작

III. 명령어 분석 및 실험 환경

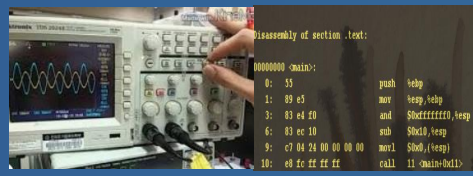


❖ 프로세서 명령어 종류

- 명령어 세트 매뉴얼 기준 총 137개 명령어
- 분기 명령어 등 제외한 40개 명령어가 주요 분석 대상

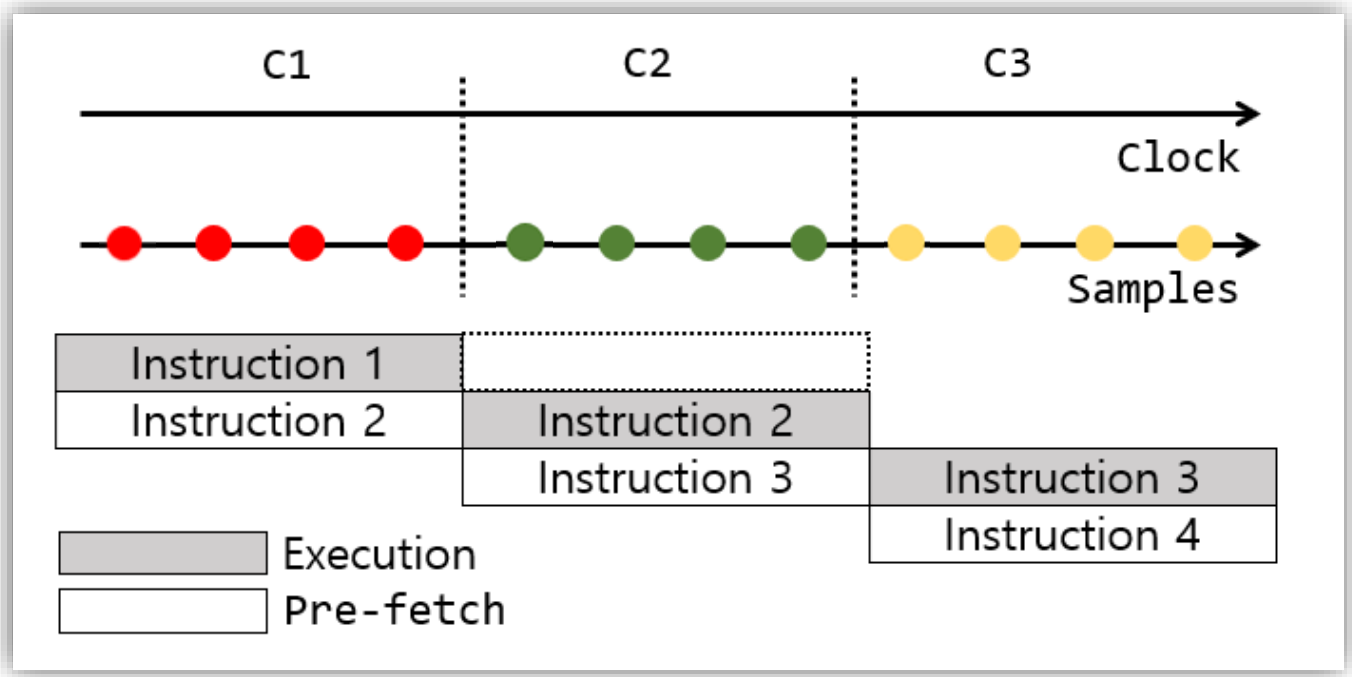
XMEGA128 명령어 셋				
ADD(✓)	ADC(✓)	ADIW(✓)	SUB(✓)	SUBI(✓)
SBC(✓)	SBCI(✓)	SBIW(✓)	AND(✓)	ANDI(✓)
OR(✓)	ORI(✓)	EOR(✓)	COM	NEG
SBR	CBR	INC(✓)	DEC(✓)	TST
CLR	SET	MUL	MULS	MULSU
FMUL	FMULS	FMULSU	RJMP(✓)	IJMP
EIJMP	JMP	RCALL	ICALL	EICALL
CALL	RET	RETI	CPSE	CP(✓)
CPC(✓)	CPI(✓)	SBRC	SBRS	SBIC
SBIS	BRBS	BRBC	BREQ(✓)	BRNE(✓)
BRCS(✓)	BRCC(✓)	BRSH	BRLO	BRMI
BRPL	BRGE(✓)	BRLT(✓)	BRHS	BRHC
BRTS	BRTC	BRVS	BRVC	BRIE
BRID	MOV(✓)	MOVW(✓)	LDI(✓)	LDS(✓)
LD(✓)	LDD(✓)	STS(✓)	ST(✓)	STD(✓)
LPM	ELPM	SPM	IN(✓)	OUT(✓)
PUSH	POP	XCH	LAS	LAC
LAT	LSL	LSR(✓)	ROL	ROR(✓)
ASR(✓)	SWAP(✓)	BSET	BCLR	SBI
CBI	BST	BLD	SEC	CLC
SEN	CLN	SEZ	CLZ	SEI
CLI	SES	CLS	SEC	CLV
SET	CLT	SEH	CLH	BREAK
NOP	SLEEP	WDR		

III. 명령어 분석 및 실험 환경

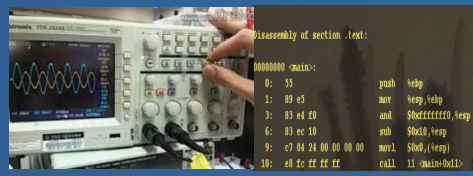


❖ 실행 명령어 코드 구성

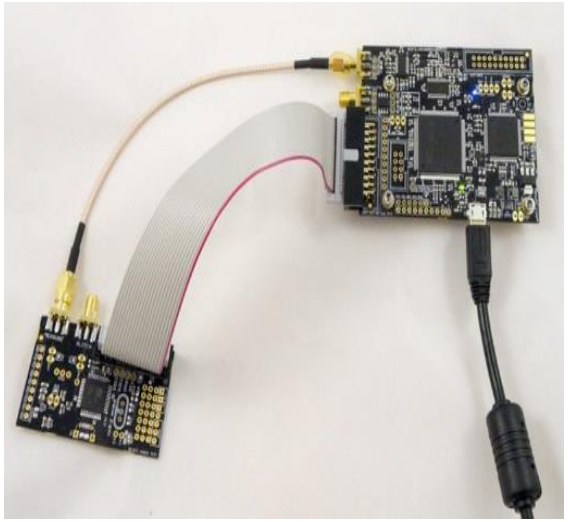
- 최대 샘플 속도 : 4 Samples/CLK(ChipWhisperer-Lite 사용시)
- 선인출(pre-fetch)과 실행(execution) 수행
- 한 클록당 명령어 실행과 다음 명령어의 선인출 수행



III. 명령어 분석 및 실험 환경

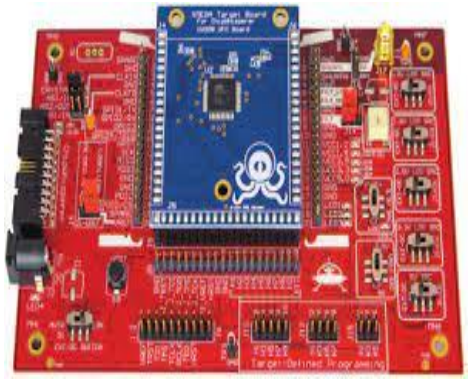


❖ 실험 장비 및 환경 고도화

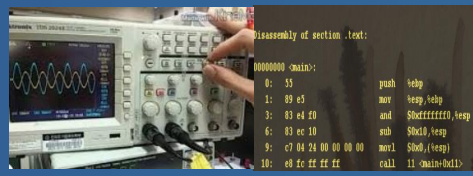


- 기본 환경: ChipWhisperer-Lite 활용
  - PC, ChipWhipserer-Lite 캡처보드, 타겟 보드
  - 1클럭 사이클당 4샘플 측정
  - 낮은 Sampling rate로 분석에 한계점 존재

- 고도화 환경: 오실로스코프를 이용한 측정
  - 타겟 보드: ChipWhisperer UFO 베이스 보드 &
    - XMEGA128 (8-bit)
    - STM32F0 (32-bit Cortex-M0)
  - 오실로스코프: LeCroy WaveRunner 8404M-MS
  - 베이스 보드 전원 공급기(5V)

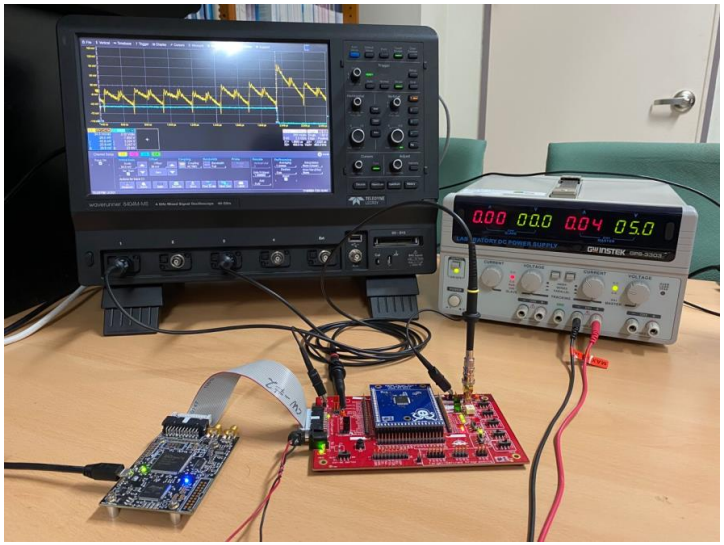


III. 명령어 분석 및 실험 환경



❖ 명령어 템플릿 측정 환경

- Oscilloscope 세팅
  - Lecroy HDO4021 200MHz bandwidth
  - Resolution: 8bit
  - Sampling rate: 기존 2.5 → 1.25GS/s
  - Invert 모드 (상하반전)



명령어 데이터 셋 구축 환경

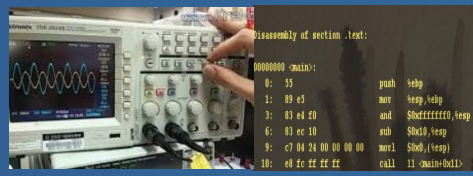
- Profiling/Test board
  - Clock cycle: 4MHz
    - 클럭당 312.5 샘플로 조정
  - 동작 전압: 3.3V

베이스 보드

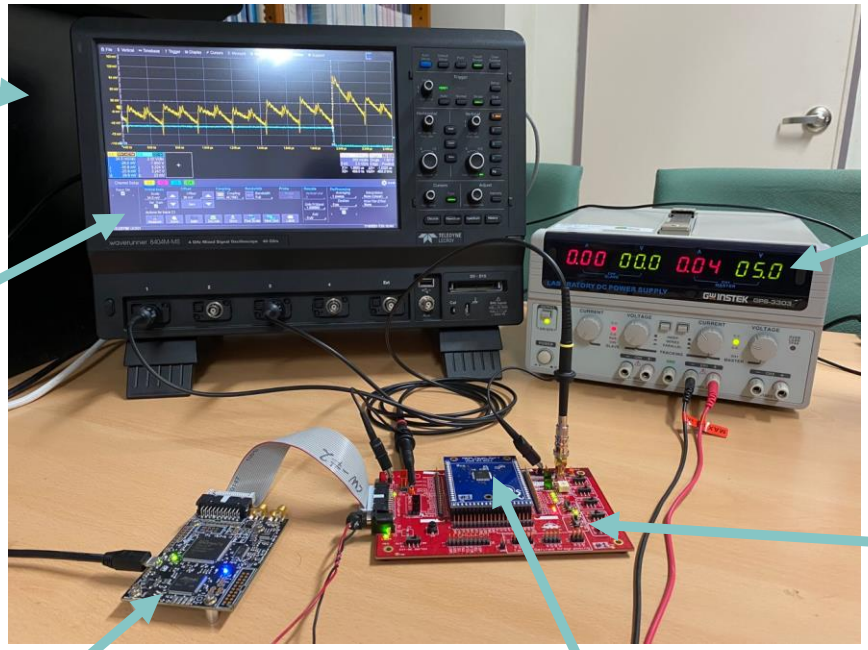
Profiling MCU

Target MCU

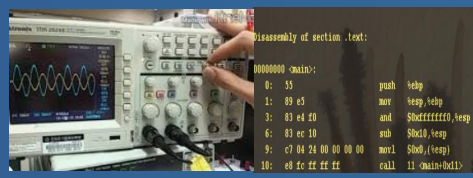
III. 명령어 분석 및 실험 환경



❖ 명령어 템플릿 측정 장비 및 환경



IV. 명령어 템플릿 구성

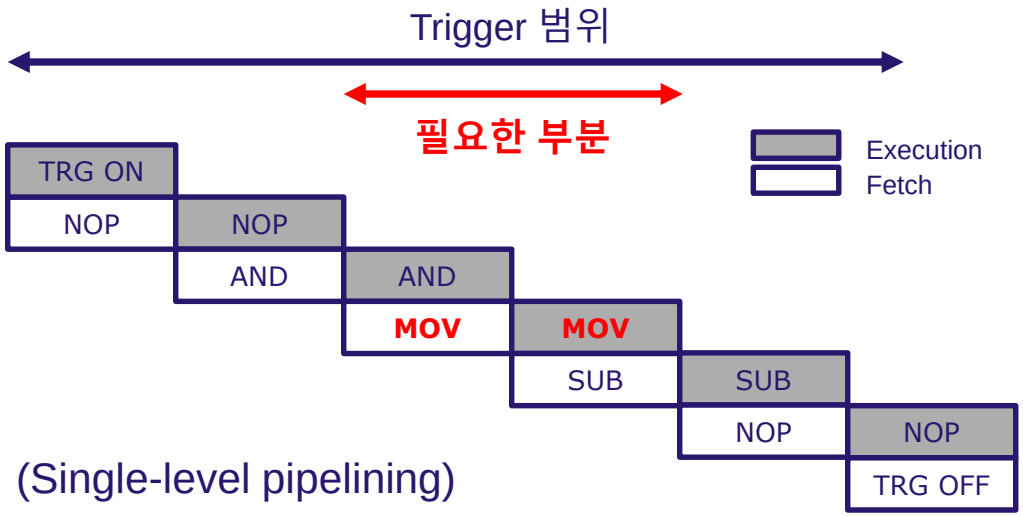


❖ 개별 명령어 파형 측정

- 특정 Opcode의 전력 측정을 위한 펌웨어 (XMEGA128)

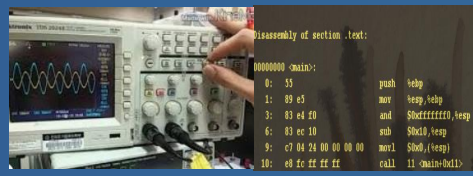
```
#define PUSH_REGS asm volatile ("PUSH R0\n\tPUSH R1\n\t ... ")
#define POP_REGS  asm volatile ("POP R31\n\tPOP R30\n\t ... ")

uint8_t execute(uint8_t* buf, uint8_t len)
{
    PUSH_REGS;
    trigger_high();
    asm volatile (
        "NOP \n\t"
        "AND r4, r27\n\t"
        "MOV r8, r19\n\t"
        "SUB r28, r2\n\t"
        "NOP \n\t"
    );
    trigger_low();
    POP_REGS;
    return 0x00;
}
```

랜덤화



(Single-level pipelining)

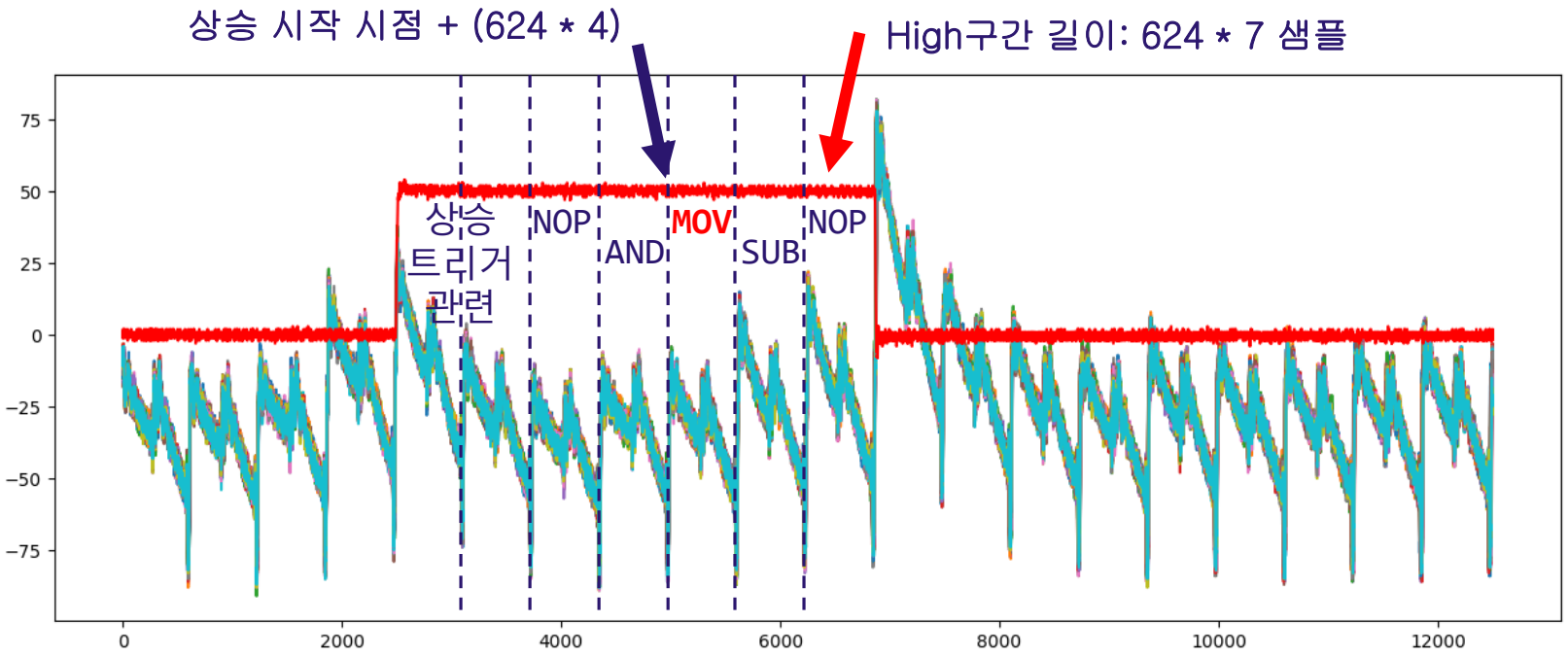
IV. 명령어 템플릿 구성



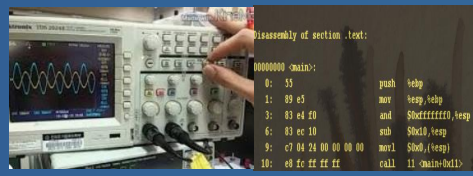
❖ 개별 명령어 파형 추출

- 트리거 신호 측정 및 활용
  - 상승 트리거 지점 + (624 * n)을 통해 위치 계산
  - 이론상 1클럭당 625샘플, 실제로는 624 샘플

```
trigger_high();
asm volatile (
     "NOP \n\t"
     "AND r4, r27\n\t"
     "MOV r8, r19\n\t"
     "SUB r28, r2\n\t"
     "NOP \n\t"
);
trigger_low();
```



IV. 명령어 템플릿 구성



❖ 템플릿 구성 방법의 제약 요소

- 측정 대상의 앞/뒤 명령어를 랜덤하게 선택함
- 측정 대상 명령어가 N개 있다고 하면 N^3개의 조합 가능
  - 100개라 가정해도 100만개의 조합 존재
- 컴파일러 생성 코드의 제한성
  - N^3개의 조합에는 컴파일러가 생성하지 않는 불필요한 코드 조합이 존재
  - 일정 규칙에 따라 특정한 패턴 코드를 반복 사용
- 랜덤하게 생성한 템플릿

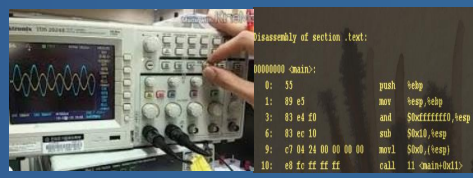
➔ 실제 코드(real code)와 템플릿의 일치성이 낮음

실제 명령어 복구가 어려움

```
trigger_high();
asm volatile (
      "NOP \n\t"
      "AND r4, r27\n\t"
      "MOV r8, r19\n\t"
      "SUB r28, r2\n\t"
      "NOP \n\t"
);
trigger_low();
```

MOV 명령어를 제외하고
Opcode 및 Operand를
랜덤하게 선택

IV. 명령어 템플릿 구성



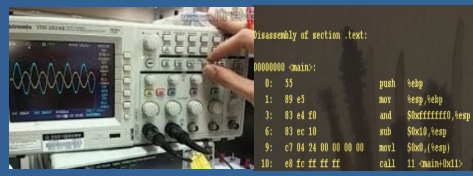
❖ 명령어 템플릿 구성의 고도화(XMEGA의 경우)

- 매뉴얼상 명령어의 종류는 100가지 이상 존재

- 컴파일러는 주요 Opcode만을 사용해 실행파일 생성
  - 사용되지 않는 Opcode는 템플릿 구성 작업에서 제외하여 경량화
  - 예> 분기 명령어만 20개 존재 -> 6개 정도만 사용

- 명령어는 특정 패턴을 반복적으로 사용
  - 분기 명령어의 앞에는 비교문, 증감연산, 합/차 연산, 비트연산 사용
  - 예> std-std-std, ldd-ldd-ldd와 같이 동일 명령어의 조합은 자주 사용됨

IV. 명령어 템플릿 구성



❖ 명령어 템플릿 구성의 고도화(XMEGA의 경우)

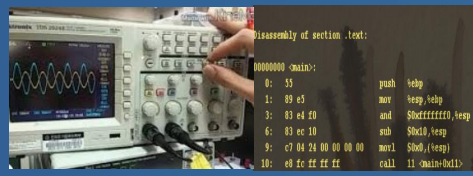
- 랜덤한 Opcode로 명령어 템플릿을 구성하는 것은 비효율적
  - Dataset에 대한 정확도는 높아도 실제 코드에 대한 복구율은 낮음
- 컴파일러가 생성하는 명령어 시퀀스 빈도수를 통계적으로 분석

➔ 이를 바탕으로 명령어 템플릿 재구성함

펌웨어 명령어 코드 예시

```
4d2:   38 27      eor     r19, r24
4d4:   3b 83      std     Y+3, r19     ; 0x03
4d6:   53 96      adiw    r26, 0x13    ; 19
4d8:   8c 91      ld      r24, X
4da:   48 27      eor     r20, r24
4dc:   4c 83      std     Y+4, r20     ; 0x04
4de:   fb 01      movw    r30, r22
4e0:   80 81      ld      r24, Z
4e2:   89 27      eor     r24, r25
4e4:   80 8b      std     Z+16, r24    ; 0x10
4e6:   81 81      ldd     r24, Z+1     ; 0x01
4e8:   9a 81      ldd     r25, Y+2     ; 0x02
4ea:   89 27      eor     r24, r25
```

IV. 명령어 템플릿 구성



❖ 명령어 시퀀스의 빈도수 분석

- 명령어 시퀀스에 대한 정의
  - 순차적으로 실행되는 3개의 Opcode 나열
  - Single-level pipeline 고려 (패치 및 실행)
    - 이전 명령어 실행 단계에서 패치
    - 다음 명령어가 패치될 때 영향을 받음

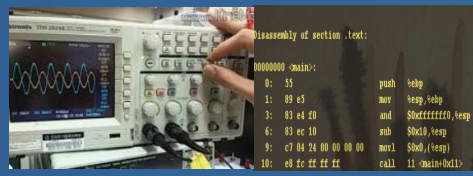
시퀀스-1	add	adc	movw
시퀀스-2	mov	adc	sbc
시퀀스-3	eor	adc	mul
시퀀스-4	sbiw	adc	eor
시퀀스-5	ld	adc	eor

- 블록 암호 알고리즘(AES, ARIA, SEED, LEA) 전체 펌웨어 분석
- avr-gcc 컴파일러의 최적화 옵션 (5종류)
  - -O0(속도 최하), -O1, -O2, -O3(속도 최고), -Os

➔ 총 20개(4개 알고리즘, 5개 옵션) 펌웨어에 적용

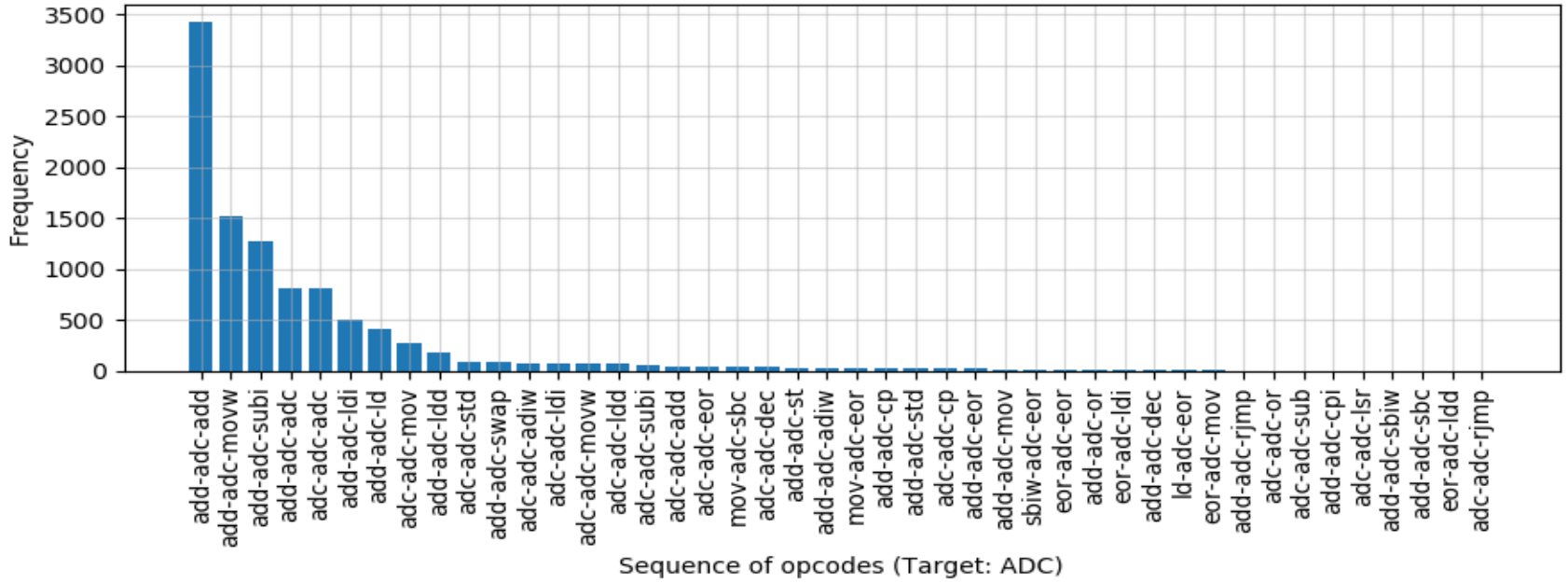
명령어 시퀀스 빈도수 분석하여 명령어 템플릿 구성

IV. 명령어 템플릿 구성

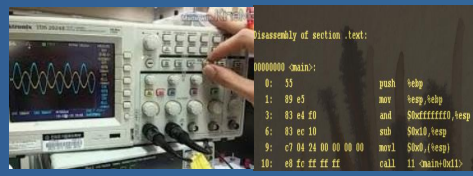


❖ 예시 : 명령어 시퀀스의 빈도수 분석

- 랜덤 템플릿 구성 시에는 총 10,000개 이상(랜덤 Opcode 2개)
- ADC 명령어는 오직 50개의 명령어 시퀀스 존재

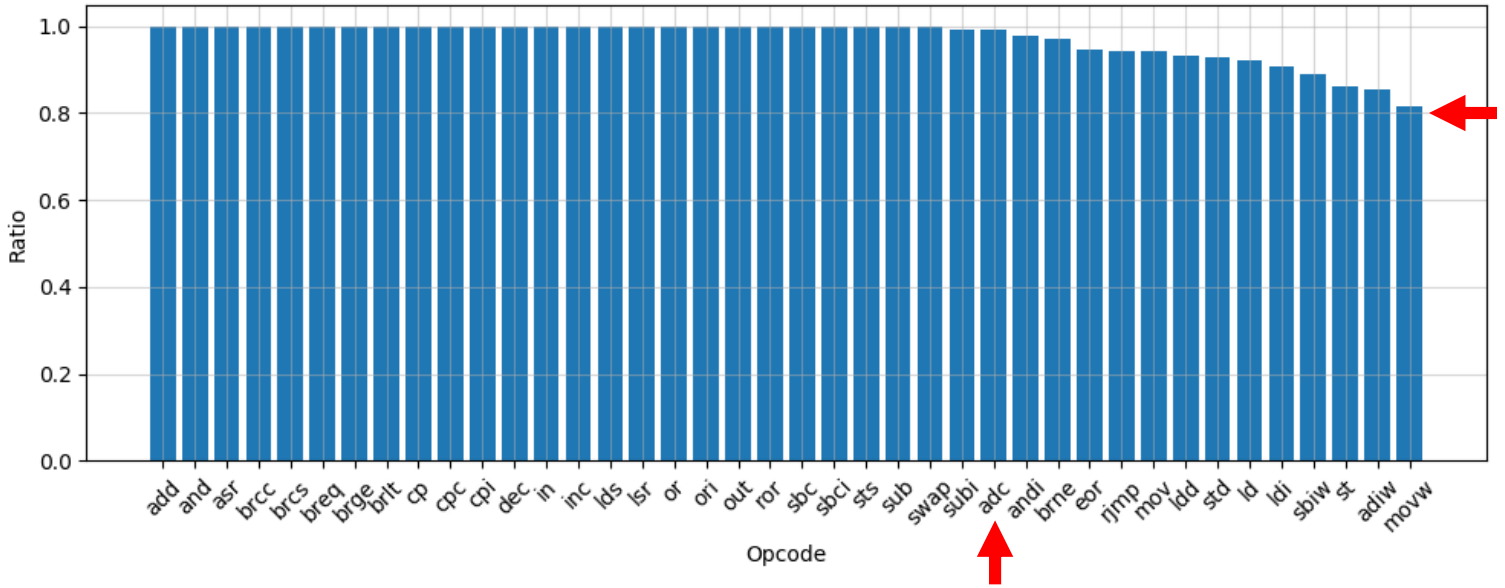


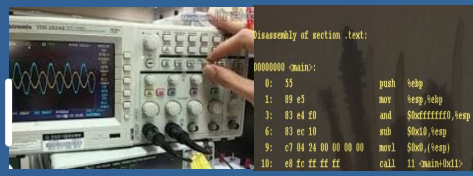
IV. 명령어 템플릿 구성



❖ 예시 : 명령어 시퀀스의 빈도수 분석

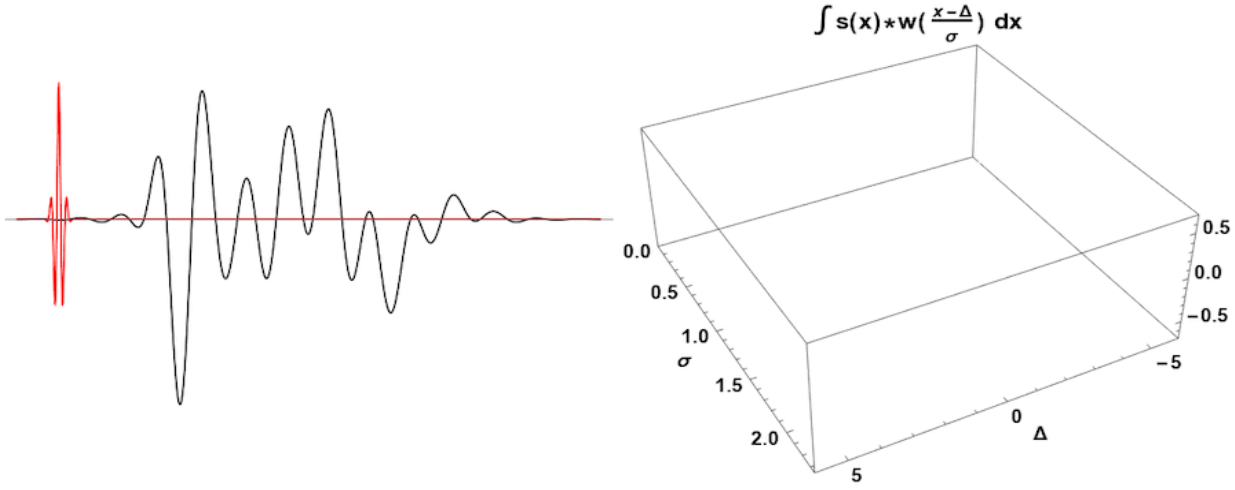
- 실제 각 명령어는 1~200개 정도의 명령어 시퀀스만 존재
- 상위 30개가 각각의 명령어 시퀀스에서 80% 이상을 차지함
- 상위 30개의 명령어 시퀀스로 템플릿 구성





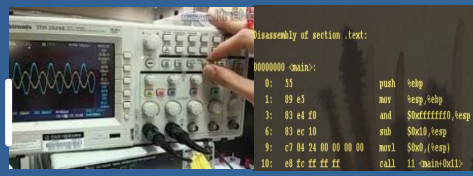
CWT를 이용한 신호 분해 및 노이즈 감쇄

- Continuous wavelet transform (CWT)
  - 시간 도메인의 1차원 신호를 시간-주파수의 2차원 도메인으로 분해
  - Mother wavelet을 이동시키며 합성곱(Convolution) 연산 수행
  - Mother wavelet 스케일과 특정 주파수 대역 추출(작은 스케일 : 고주파)
  - Mexican hat function을 Mother wavelet로 사용



https://en.wikipedia.org/wiki/Continuous_wavelet_transform

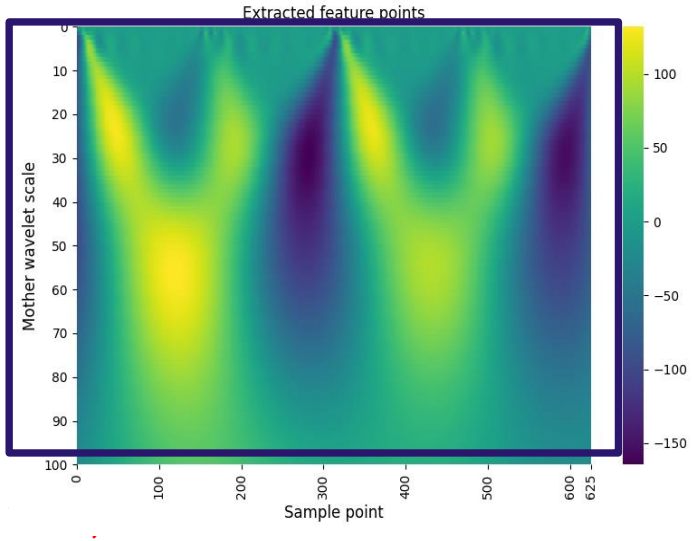
V. 머신 러닝 기법 기반 명령어 분류기



❖ 연속 웨이블릿 변환의 스케일 최적화

- Mother wavelet 스케일 매개변수 : 1 ~ 1000 (10 단위, 100개)
- 클럭당 624샘플 → 클럭당 62,400샘플로 분해
  - 고주파 대역: 전기적 잡음
  - 저주파 대역: 잡음이 제거 + 정렬된 정보

스케일 최적화 CWT 결과



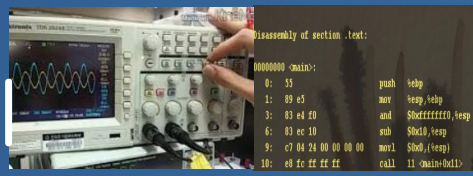
- 스케일 최적화

➔ 기존 :1에서 1,000까지 100개 스케일 사용

➔ 개선 :1에서 156까지 100개를 사용

(2클럭 주파수 성분만 추출 : 실제 코드 복구율

V. 머신 러닝 기법 기반 명령어 분류기



❖ KL-Divergence 기반 명령어 특징 추출

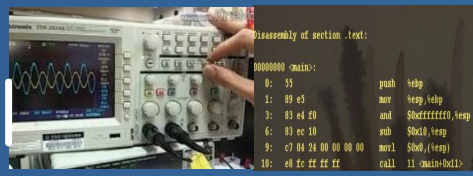
- Kullback–Leibler divergence (KLD)
  - 두 확률 분포 p, q의 차이를 나타냄 → KLD(p, q)
  - 상대 엔트로피 (relative entropy)하고도 함
    - KLD(p, q) = p 분포에 대해, 이를 근사하는 q 분포를 사용해 샘플링 했을 때 발생하는 **엔트로피의 차이**
    - 이산 확률 분포의 정보 Entropy: $-\sum_k \boldsymbol{p_k} \cdot \log_2 \boldsymbol{p_k}$

❖ 분류에 사용될 수 있는 명령어 특징?

- A 조건 : 같은 Opcode에 대해 낮은 KLD 지점
- B 조건 : 다른 Opcode에 대해 높은 KLD 지점

2가지 모두 만족시키는 지점

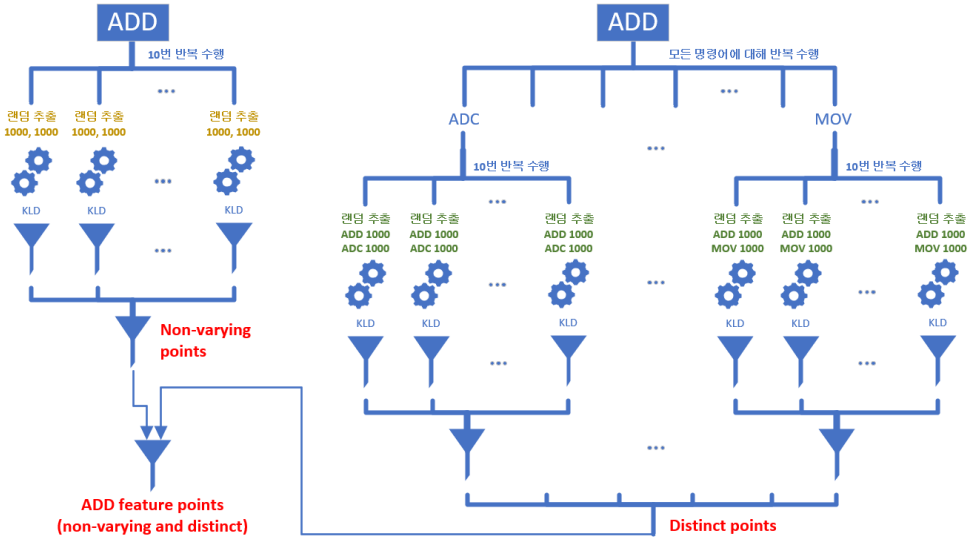
V. 머신 러닝 기법 기반 명령어 분류기



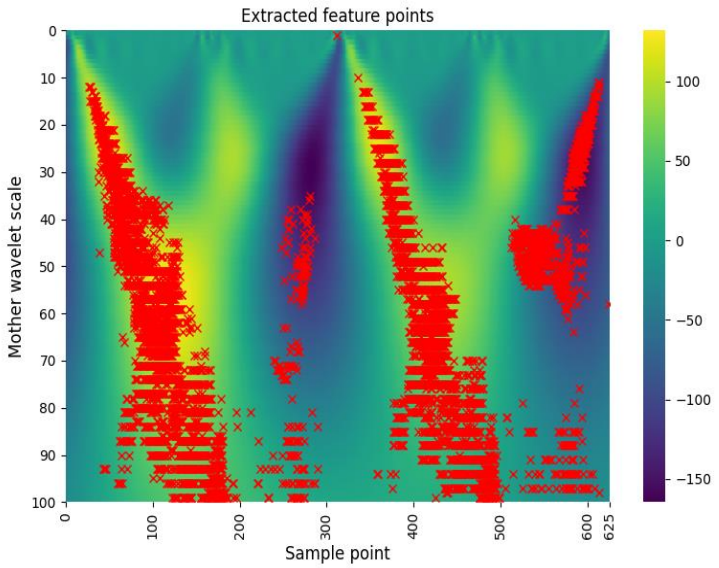
❖ 데이터셋 구축 및 특징 추출

- Kullback-Leibler Divergence 기반 명령어 특징 추출
  - CWT 적용 : 312.5샘플/CLK *2 CLK *100 스케일 =62,500 샘플
  - KLD를 적용하여 40개 명령어에서 전체 4,696개 특징점 추출

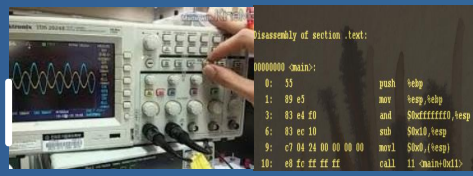
KLD 기반 명령어 특징 추출 절차



추출된 특징 포인트

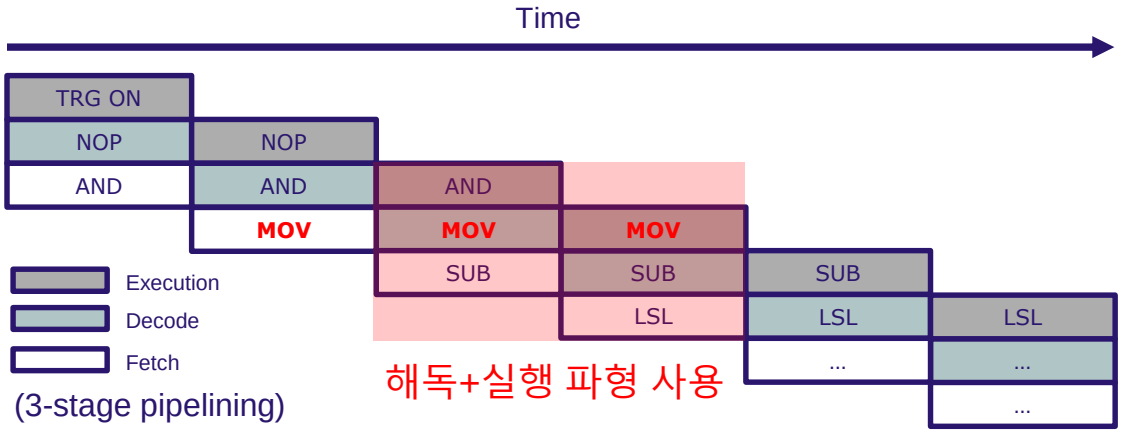


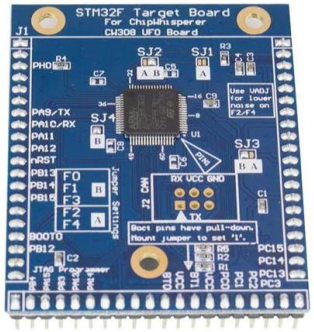
V. 머신 러닝 기법 기반 명령어 분류기



32비트 STM32F0의 명령어 복구

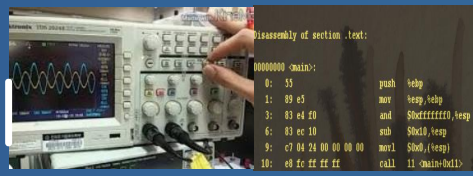
- ARM Cortex-M0 32비트 프로세서 탑재 (ARMv6-M아키텍처)
- Thumb 명령어 셋의 주요 명령어 23개 복구 실험
- 3단계 파이프라인 구조
  - 인출(Fetch) – 해독(Decode) – 실행(Execution)
  - Fetch 시 명령어간 전력 차이가 작음





STM32F0 MCU를 탑재한 타겟 보드

V. 머신 러닝 기법 기반 명령어 분류기

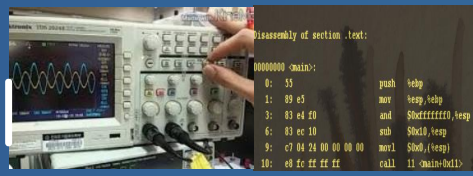


❖ STM32F0 프로세서 명령어 종류

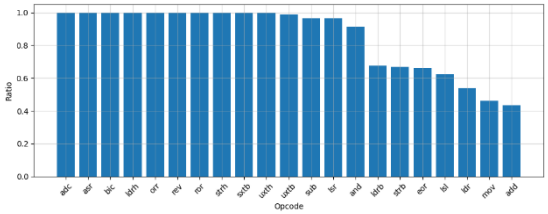
- 명령어 세트 매뉴얼 기준 총 56개 명령어
- 분기 명령어 등 제외한 23개 명령어가 주요 분석 대상

16비트 Thumb 명령어 셋				
ADC(√)	ADD(√)	ADR	AND(√)	ASR(√)
B	BIC(√)	BLX	BKPT	BX
CMN	CMP(√)	CPS	EOR(√)	LDM
LDR(√)	LDRH(√)	LDRSH	LDRB(√)	LDRSB
LSL(√)	LSR(√)	MOV(√)	MVN	MUL
NOP	ORR(√)	POP	PUSH	REV(√)
REV16	REVSH	ROR(√)	RSB	SBC
SEV	STM	STR	STRH(√)	STRB(√)
SUB(√)	SVC	SXTB(√)	SXTH	TST(√)
UXTB(√)	UXTH(√)	WFE	WFI	YIELD
32비트 Thumb 명령어 셋				
BL	DSB	DMB	LSB	MRS
MSR				

V. 머신 러닝 기법 기반 명령어 분류기



32비트 STM32F0의 명령어 복구



- 명령어 템플릿 구성?
  - 3단계 파이프라인을 고려해 1개의 명령어 뒤에 추가
  - 명령어 빈도수 분석 수행 (4개의 명령어 단위)

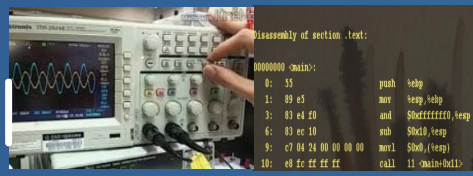
```
trigger_high();
asm volatile (
     "NOP \n\t"
     "SUB r0, #12\n\t"
     "ADC r2, r4\n\t"
     "LSR r6, r3\n\t“
     "MOV r0, #153\n\t"
     "NOP \n\t"
);
trigger_low();
```

- 명령어 템플릿 구성(hex) 작업을 제외한

  모든 작업은 XMEGA 경우와 동일 (자동화)

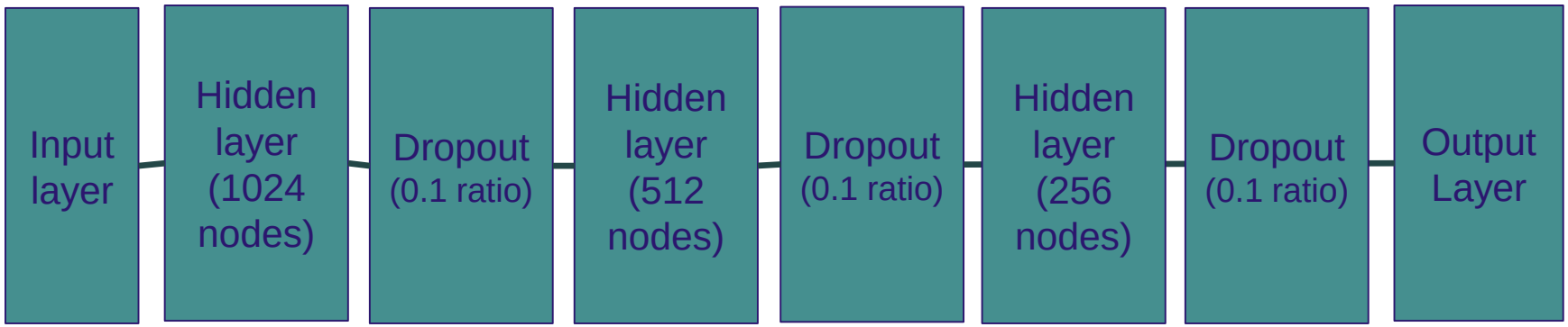
- 학습 및 테스트 데이터의 구성(파형당 2,343개 샘플)
  - 학습용 데이터 3만6천개
    - 21개의 Opcode 대상
    - 10~25개의 명령어 시퀀스
    - 각각 100개의 파형
  - Cross-board 검증 데이터 3천개
    - 21개의 Opcode 대상
    - 5개의 명령어 시퀀스
    - 각각 25개의 파형

V. 머신 러닝 기법 기반 명령어 분류기

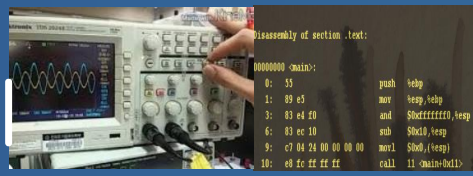


❖ 딥러닝 모델인 MLP(Multi-Layer Perceptron)

- 1024, 512,256 개의 뉴런으로 구성된(혹은10개)의 은닉계층
- 활성화 함수 : ReLu 함수 사용
- 손실 함수 : Binary 또는 Categorical crossentropy
- 최적화 : Adam optimizer (0.001 learning rate)
- 과적합(overfitting) 방지 ➔ Dropout(0.1 비율) 적용



V. 머신 러닝 기법 기반 명령어 분류기

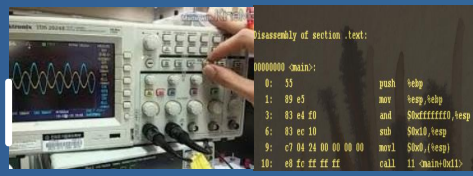


❖ 딥러닝을 이용한 명령어 분류

- 학습 및 테스트 데이터의 구성(파형당 4,696개 샘플)
  - 학습용 데이터 12만개
    - 40개의 Opcode 대상
    - 30개의 명령어 시퀀스
    - 각각 100개의 파형
  - Cross-board 검증 데이터 4천개
    - 40개의 Opcode 대상
    - 5개의 명령어 시퀀스
    - 각각 20개의 파형

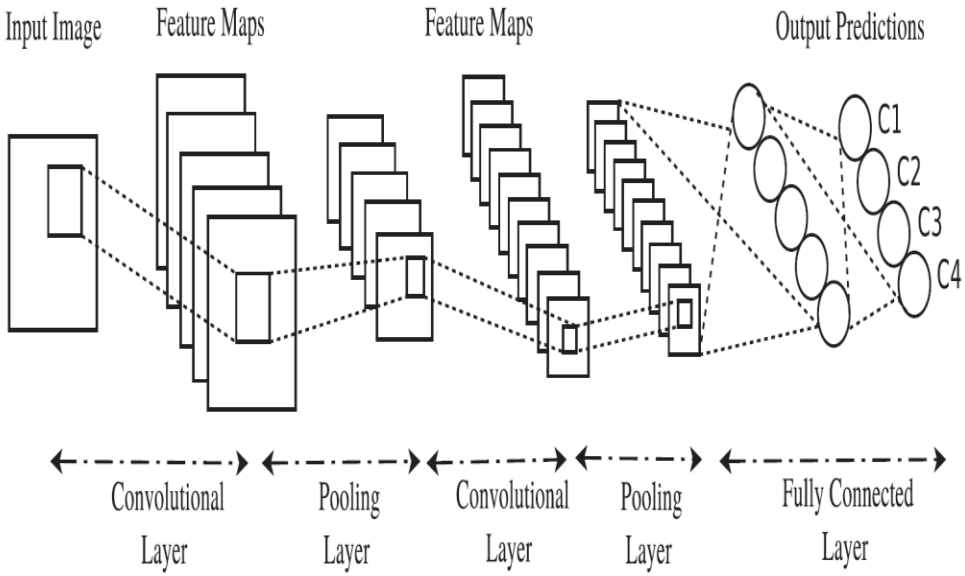
	XMEGA128		STM32F0	
특징 포인트	4,696		2,343	
분석 대상 Opcode	40		21	
각 Opcode별 펌웨어 수	30	5	10~25	5
각 펌웨어별 파형 수	100	25	100	25
총 데이터 수	120,000	5,000	36,000	2,625

V. 머신 러닝 기법 기반 명령어 분류기



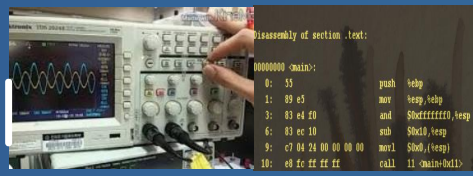
CNN 기반의 명령어 수준 역어셈블러

- 입력 정보에 대한 특징 추출을 위해 마스크(필터) 도입
- 활성화 함수 : ReLU, Softmax
- 과적합 : Dropout(비율 : 0.25)



Layer	Shape/Kernel/ Node/Dropout ratio	Activation
(Input)	39x8x1	N/A
Conv2D	32 (7x4)	ReLU
Conv2D	32 (5x3)	ReLU
Conv2D	32 (4x2)	ReLU
Conv2D	32 (2x2)	ReLU
Dropout	0.25	N/A
(Input)	9,984	N/A
Dense	500	ReLU
Dropout	0.25	N/A
Dense	61	Softmax

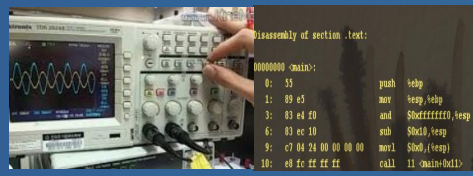
V. 머신 러닝 기법 기반 명령어 분류기



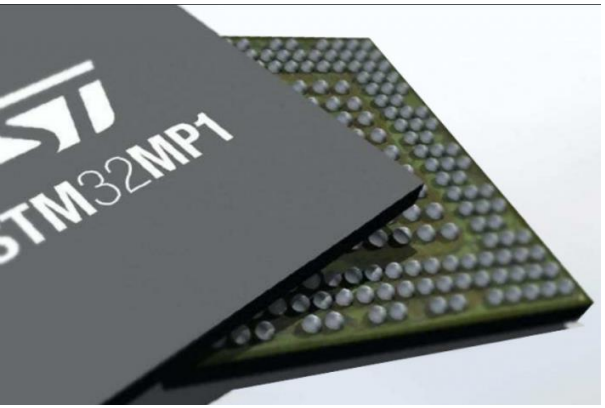
❖ 딥러닝 및 기계학습을 이용한 명령어 분류

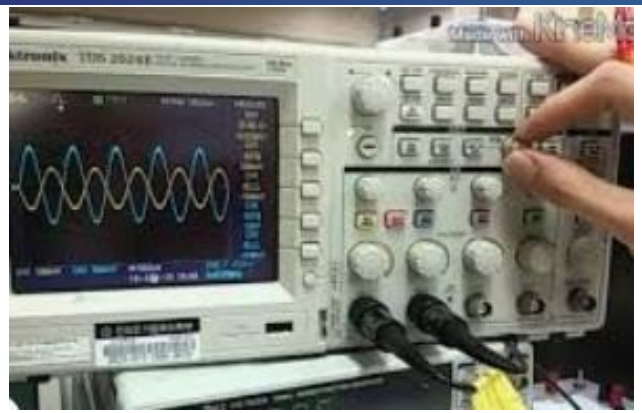
	XMEGA128		STM32F0	
	Validation	Cross-board	Validation	Cross-board
KNN (k=3)	90.8%	64.2%	98.8%	84.6%
Random Forest	91.3%	59.7%	98.7%	81.6%
CNN	66.7%	54.4%	84.8%	74.2%
MLP(shallow-3층)	76.18%	70.6%	98.2%	93.6%
MLP(deep-10층)	**91.9%**	**77.0%**	**98.6%**	**96.5%**

VI. 맺 음 말



- 하드웨어 실행 코드 역어셈블러의 필요성
- 머신 러닝을 이용한 역어셈블러 구현 연구
- 역어셈블러의 연구 방향
  - 실제 구동 환경하에서 명령어 템플릿 구성
  - 명령어에 대한 POI 설정 및 신호 측정
  - 전력 신호 데이터에 대한 사전 신호 처리
  - 머신 러닝 모델의 최적화 및 명령어 복구율
  - 역어셈블러의 다양성과 실제 환경을 고려한 연구가 필요





```
Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 e4 f0                and    $0xfffffff0,%esp
   6:   83 ec 10                sub    $0x10,%esp
   9:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  10:   e8 fc ff ff ff          call   11 <main+0x11>
```

감사 합니다.

Q & A