AVR 8bit에서 격자 기반 PQC 알고리즘 구현 가이드

PQC Algorithm Implementation Guide

on AVR 8bit

김동영

오원영

우주 갈끄니까~

문경태

권보연

박정식

- 목 차 -

제 1 장 개요

- 이 문서에서는 AVR8bit에서 SMAUG-T를 구현하면서 적용된 구현 방법론을 분석하면서, TiMER 기반 PQC 구현에 도움이 될 수 있는 가이드 라인을 제공한다.

제 2 장 AVR8bit 및 개발 환경 구현

제 2.1 절 AVR8bit

■ AVR MCU

☐ Atmel의 AVR(Advanced Virtual RISC)은 저가형 프로세서로, 2016년 Microchip에 인수되었었다. CPU, ROM, RAM 등 다양한 하드웨어를 통합하고 ISP를 지원해 PC에서 쉽게 프로그래밍할 수 있어 인기 있다. 저렴한 가격과 강력한 전기적 특성으로 대학교 강의 및 졸업작품에 자주 활용된다.

☐ AVR은 고성능이 필요 없는 제어 장치에 적합하며, Arduino Uno와 같은 제품에 사용된다. 특히 ATMega128은 가격, 성능, 확장성 면에서 널리 쓰이며, I/O 핀이 최대 40mA 전류를 처리할 수 있어 설계가 간단하다.

☐ 다만, 8비트 마이크로컨트롤러로서 성능이 제한적이며 고성능이 필요한 분야에서는 ARM 기반 MCU가 선호된다. 구조가 단순하고 OS 없이 실행 가능해 단순 제어 분야에서 여전히 널리 쓰이고 있다.

■ ATmega 시리즈

☐ **AVR 아키텍처**는 RISC 기반의 8-bit 명령어 세트를 사용하며, ATmega 시리즈는 AVR 제품군 중 고성능 범용 마이크로컨트롤러를 대표합니다. ATmega는 CPU, ROM, RAM, 플래시 메모리, ADC, DAC, GPIO 등 다양한 기능을 통합하여 소형 및 저비용 시스템에 적합합니다. 특히, 하버드 아키텍처 기반으로 프로그램 메모리와 데이터 메모리가 분리되어 효율적인 동작이 가능합니다.

- ATmega 시리즈는 **유연한 전원 관리**를 지원하며, 8-bit 프로세서로는 드물게 **강력한 전기적 특성**을 가지고 있어 I/O 핀당 최대 40mA의 전류를 처리할 수 있습니다. 단순한 구조와 낮은 비용으로 인해 다양한 임베디드 시스템에서 널리 활용됩니다.
- 그러나 ATmega 시리즈는 FPU가 없고, 부동소수점 연산은 소프트웨어적으로 처리해야 하므로 복잡한 계산에는 적합하지 않습니다. 고성능이 필요한 애플리케이션에서는 ARM Cortex-M 시리즈 같은 고급 MCU가 선호되지만, ATmega는 단순 제어, 교육, 프로토타이핑과 같은 용도에서 여전히 강력한 경쟁력을 유지하고 있습니다.

제 2.2 절 ATmel Studio 개발 환경 및 구축 가이드

■ ATmel Studio 설치

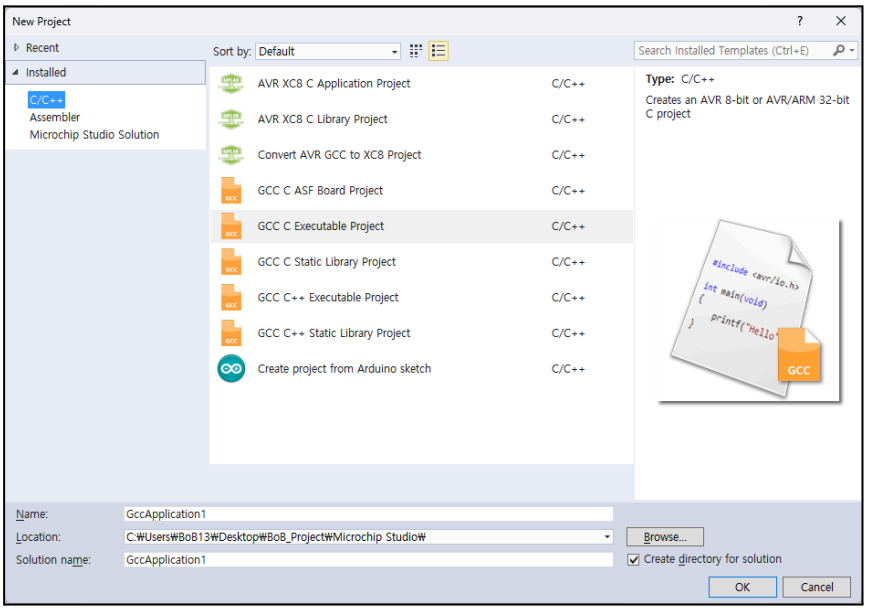
- **Atmel Studio**는 Microchip Technology에서 제공하는 통합 개발 환경(IDE)으로, AVR 및 SAM 계열의 마이크로컨트롤러 개발을 지원합니다. Atmel Studio는 **C/C++ 프로그래밍**과 **어셈블리 언어**를 사용하여 펌웨어를 작성할 수 있으며, **AVR 및 SAM 디바이스에 대한 디버깅**과 **시뮬레이션** 기능을 제공합니다.

- ATmel Studio를 설치하기 위해서는 다음 링크로 접속한다
  - https://www.microchip.com/en-us/tools-resources/develop/microchip-studio
- Michrochip Studio for AVR and SAM Devices-Offline Installer과 Michrochip Studio for AVR and SAM Devices-Web Insteller 두가지 방법이 있는데 상황에 맞게 사용하여 다운로드 하면된다.
- 설치 파일을 실행하면 설치가 진행되며, 특별한 추가 설정 없이 설치를 완료한다.

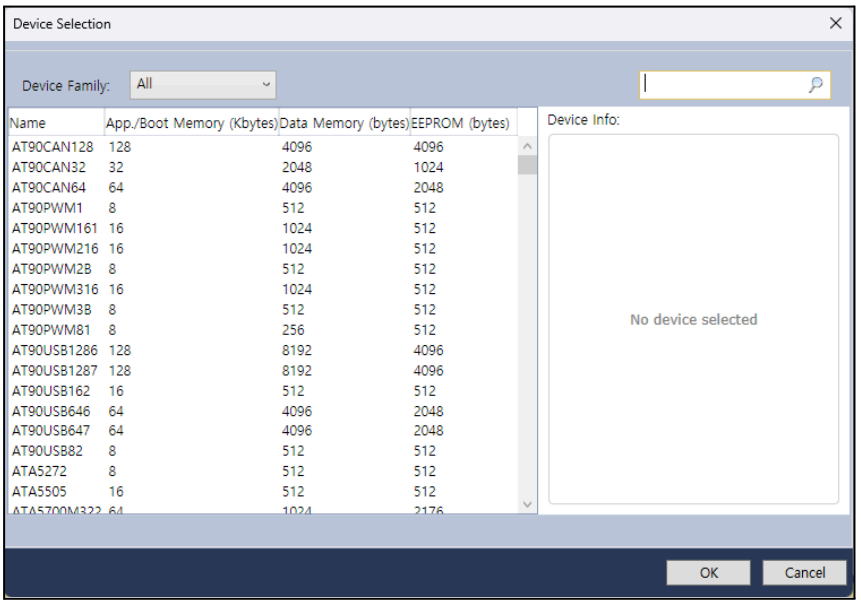
■ 보드 선택 및 기본 설정

☐ ATmel Studio를 실행한 뒤, File - New - Project 를 선택한다.

☐ 나타나는 팝업에서 사용할 언어와 제작할 프로그램을 선택한다.



☐ 다음 팝업에서 사용할 칩셋을 선정한다.



☐ 칩을 선택후에는 메인 프로그램이 생기므로 원하는 프로그램을 작성하면된다.

제 2.3 절 프로그램 빌드 및 포팅 방법

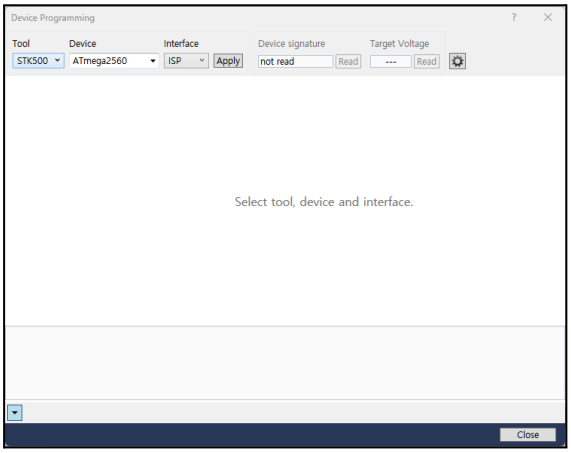
■ 프로그램 빌드

☐ Build의 Build Solution을 사용하면 output Files에 eep, elf, hex, lss, map, srec 파일들이 생긴다.

■ 프로그램 포팅

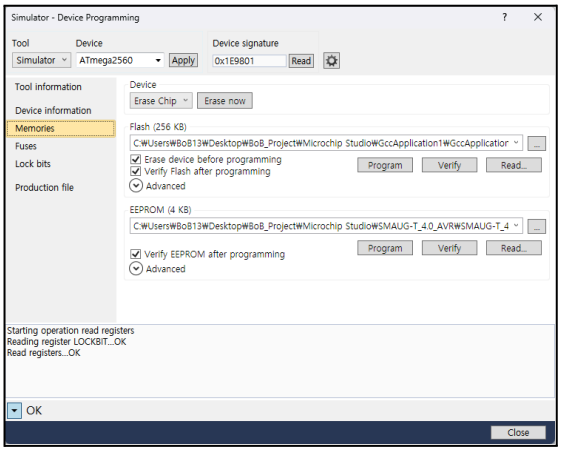
☐ 본 프로젝트에서는 ISP 케이블을 사용하여 포팅을 진행했다.

☐ Tools - Device Programming을 누르면 팝업이 뜬다.



☐ Apply를 눌러서 칩셋을 적용시킨후 Memories에 들어가서 상단 Flash의 기본 설정 Program을 누르면 Memory누르면 기존 FlashMemory의 데이터를 삭제하고 새로 쓴뒤 검증절차까지 한다.

☐



☐ 보통 Porting 하는과정에서의 에러는 잘 안나는데 시리얼 포트와 ISP 케이블을 동시에 연결하고 있으면 문제가 생기므로 하나의 포트만 사용하는것을 권장한다.

제 3 장 최적화 적용된 구현 기술 분석 및 가이드

제 3.1 절 keccak-f1600

■ keccak-f1600 기반의 SHA-3 and SHAKE

☐ FIPS-202에 정의된 SHA-3는 keccak 순열함수 (keccak-f1600)을 기반으로 설계된 암호학적 해시함수이다. 또한, 같은 문서에 정의된 SHAKE는 확장 가능한 출력함수로 SHA-3와 동일하게, keccak 순열함수를 사용한다.

■ SHA-3 및 SHAKE의 AVR8bit 구현

☐ SHA-3와 SHAKE의 개발자들이 제공하는 XKCP 라이브러리(https://github.com/XKCP/XKCP)에는 SHA-3와 SHAKE의 핵심 함수인 keccak 순열의 최적화 구현물들이 각 아키텍처마다 존재한다.

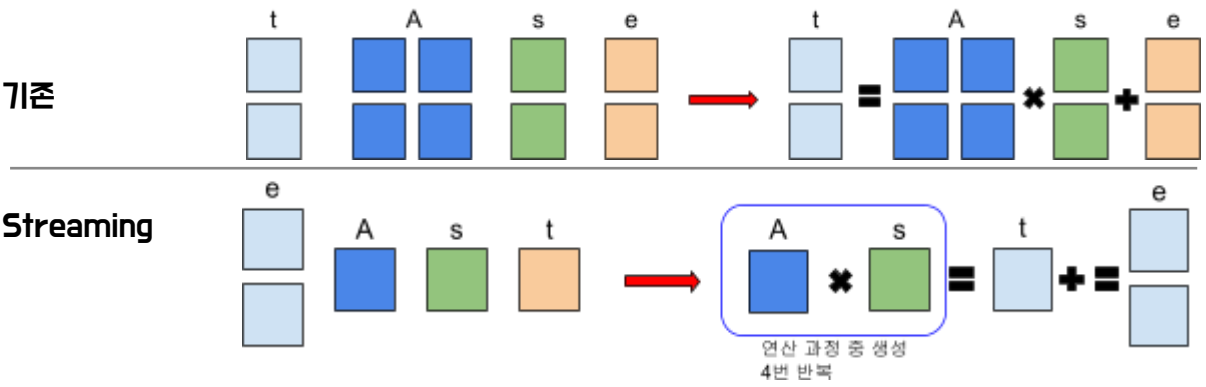
☐ 본 프로젝트에서는 SHA-3, SHAKE 코드를 사용하는 방법은 다음과 같다. 테스트 함수를 기준으로 서술한다.

☐ AVR8bit에서 사용하기 위한 KecckaP-1600-SnP.h 헤더를 사용하고 kecckaP-1600-compact.s, KeccakP-1600-avr8-fast.s 중에 상황에 맞게 적절히 사용하면 된다.(https://github.com/XKCP/XKCP/tree/master/lib/low/KeccakP-1600/AVR8)

제 3.2 절 Streaming coefficient strategy

■ Streaming

☐ SMAUG-T의 구현은 계수단위로 공개행렬을 생성하여 할 수 있다. 비밀 벡터 또한 계수 단위로 생성하여 구현 할 수 있다. 하기 그림은 계수에 대한 스트리밍 구현 방안의 도식도이다.



☐ 기존 방식은 미리 변수를 선언해두고 값을 생성한 후 필요한 값들 간의 연산으로 메모리를 많이 사용한다. 반면 Streaming 방식은 작은 크기의 메모리를 선언한 후 필요한 연산 과정에서 값을 생성하기 때문에 메모리 사용량을 대폭 줄일 수 있다.

Reference SMAUG-T pk 생성

```
void genPubkey(public_key *pk, const secret_key *sk,
               const uint8_t err_seed[CRYPTO_BYTES]) {
    genAx(pk->A, pk->seed);

    memset(&(pk->b), 0, sizeof(uint16_t) * LWE_N);
    // Initialized at addGaussian, Unnecessary
    genBx(&(pk->b), pk->A, sk, err_seed);
}

void genAx(polyvec A[MODULE_RANK], const uint8_t seed[PKSEED_BYTES])
{
    unsigned int i, j;
    uint8_t buf[PKPOLY_BYTES] = {0}, tmpseed[PKSEED_BYTES + 2];
    memcpy(tmpseed, seed, PKSEED_BYTES);
    for (i = 0; i < MODULE_RANK; i++) {
        for (j = 0; j < MODULE_RANK; j++) {
            tmpseed[32] = i;
            tmpseed[33] = j;
            shake128(buf, PKPOLY_BYTES, tmpseed, PKSEED_BYTES + 2);
            bytes_to_Rq(&A[i].vec[j], buf);
        }
    }
}

void genBx(polyvec *b, const polyvec A[MODULE_RANK], const polyvec *s,
           const uint8_t e_seed[CRYPTO_BYTES]) {
    // b = e
    addGaussianErrorVec(b, e_seed);


    // b = -a * s + e
    matrix_vec_mult_sub(b, A, s);
}
```

Streaming SMAUG-T pk 생성

```
void matacc(polyvec* pkb, uint8_t* sk, const uint8_t* seed ){
    unsigned int i, j, k;
    poly A, S, res;
    uint8_t h_buf[PKPOLY_BYTES] = { 0 };
    uint8_t errseed[CRYPTO_BYTES] = { 0 };
    uint8_t pkseed[PKSEED_BYTES + 2];

    memcpy(errseed, seed, CRYPTO_BYTES);
    memcpy(pkseed, seed + CRYPTO_BYTES, PKSEED_BYTES);

    //e를 생성해서 pkb에 저장
    addGaussianErrorVec(pkb, errseed);

    for (i = 0; i < MODULE_RANK; i++) {
        memset(&res, 0, sizeof(poly));

        for (j = 0; j < MODULE_RANK; j++) {
            pkseed[32] = i;
            pkseed[33] = j;

            //연산 전 A 생성
            shake128(h_buf, PKPOLY_BYTES, pkseed, PKSEED_BYTES + 2);
            bytes_to_Rq(&A, h_buf);
            for (k = 0; k < LWE_N; ++k){
                A.coeffs[k] = A.coeffs[k] >> (16 - LOG_Q);

                if(i == 0){
                    //처음 S 생성 후 배열에 packing
                    genSx_poly(&S, seed, j);
                    Sx_to_bytes(sk + SKPOLY_BYTES * j, &S);
                }else{
                    //두번째 부터는 생성이 아니라 unpaking
                    bytes_to_Sx(&S, sk+ j * SKPOLY_BYTES);
                }

                //Polynomial 누적 곱셈 연산
                poly_mul_acc(A.coeffs, S.coeffs, res.coeffs);
            }

            for (j = 0; j < LWE_N; ++j)
                res.coeffs[j] <<= (16 - LOG_Q);

            //poly sub
            for (j = 0; j < LWE_N; j++)
                pkb->vec[i].coeffs[j] = pkb->vec[i].coeffs[j] - res.coeffs[j];
        }
    }
```

제 3.3 절 Polynomial multiplication strategy

■ 다항식 곱셈

☐ 다항식 곱셈은 격자 기반 암호에서 핵심적인 연산으로, 효율적인 구현이 전체 성능에 큰 영향을 미칩니다. AVR8bit 환경에서 다항식 곱셈을 최적화하기 위해 다양한 알고리즘을 사용할 수 있으며, 이 절에서는 다음의 전략과 기술을 소개한다.

■ Naive Multiplication

☐ 단순한 다항식 곱셈으로, 두 다항식의 각 계수를 직접 곱한 후 결과를 누적한다.

☐ 다항식 차수가 매우 낮거나 메모리 제약이 심할 때 사용한다. 연산량 $O(n^2)$

■ Karatsuba

☐ Karatsuba는 다항식을 분할 정복(divide-and-conquer) 기법으로 곱하는 알고리즘이다. 두 n-차 다항식을 A(x), B(x)라 할 때, 각각을 두 개의 n/2-차 다항식으로 분할한다.

☐ n 〉 64 정도의 중간 크기 다항식에서 성능 최적화를 위해 유용하다. 연산량 $O(n^{1.585})$

■ Toom-Cook

☐ Karatsuba의 확장 버전으로, nnn-차 다항식을 kkk-개의 더 낮은 차수 다항식으로 분할하여 계산한다. Toom-Cook-3way의 경우 A(x),B(x)A(x), B(x)A(x),B(x)를 3개의 부분 다항식으로 나누어 곱하고, 병합한다.

☐ n 〉 128 이상의 고차 다항식에서 유용하다. $O(n^{1.404})$

■ Striding Toom-Cook

☐ Toom-Cook을 Striding 방식으로 최적화하여, 분할된 부분 다항식의 곱셈을 순차적으로 계산하고 병합한다. Toom-Cook 보다 메모리를 적게 사용한다.

☐ n 〉 128 이상의 고차 다항식에서 유용하다. $O(n^{1.404})$

☐ 본 프로젝트에서는 AVR8bit에 최적화 하는데 Striding Toom-Cook를 사용했다.

■ NTT

☐ 푸리에 변환(FFT)의 정수 연산 버전으로. 모듈로 qqq에 대한 다항식 곱셈을 효율적으로 수행한다.

☐ n 〉 256 이상의 매우 큰 차수 다항식에서 좋은 성능을 보인다. 연산량 $O(n \log n)$

제 3.4 절 Modify bit operation logic

■ 32bit 비트 연산

☐ SMAUG-T의 32bit 연산은 8비트 아키텍처의 한계. 컴파일러의 코드 생성 문제. 캐리 비트 처리의 복잡성 때문입니다. 이를 해결하려면 8비트 또는 16비트 단위로 연산을 분할하거나 어셈블리 코드를 활용하는 것이 가장 효과적이다.

Reference SMAUG-T 32bit 연산	SMAUG-T AVR8bit 최적화 비트 연산

void Rp2_to_bytes(uint8_t bytes[CTPOLY2_BYTES], const poly *data) {
  memset(bytes, 0, sizeof(uint8_t) * CTPOLY2_BYTES);
  size_t b_idx = 0;
 	
  for (size_t i = 0; i < LWE_N; i += 8) {
  uint32_t temp = 0;
 	
  for (int j = 0; j < 8; ++j) {
  temp |= ((data->coeffs[i + j] & 0x07) << (3 * (7 - j)));
  }
 	
  bytes[b_idx++] = (temp >> 16) & 0xFF;
  bytes[b_idx++] = (temp >> 8) & 0xFF;
  bytes[b_idx++] = temp & 0xFF;
  }
}
	void Rp2_to_bytes(uint8_t bytes[CTPOLY2_BYTES], const poly *data) {
	  memset(bytes, 0, sizeof(uint8_t) * CTPOLY2_BYTES);
	  size_t b_idx = 0;
	
	  for (size_t i = 0; i < LWE_N; i += 8) {
	  bytes[b_idx] |= (data->coeffs[i] & 0x07) << 5;
	  bytes[b_idx] |= (data->coeffs[i + 1] & 0x07) << 2;
	  bytes[b_idx] |= (data->coeffs[i + 2] & 0x06) >> 1;
	  b_idx++;
	
	  bytes[b_idx] |= (data->coeffs[i + 2] & 0x01) << 7;
	  bytes[b_idx] |= (data->coeffs[i + 3] & 0x07) << 4;
	  bytes[b_idx] |= (data->coeffs[i + 4] & 0x07) << 1;
	  bytes[b_idx] |= (data->coeffs[i + 5] & 0x04) >> 2;
	  b_idx++;
	
	  bytes[b_idx] |= (data->coeffs[i + 5] & 0x03) << 6;
	  bytes[b_idx] |= (data->coeffs[i + 6] & 0x07) << 3;
	  bytes[b_idx] |= (data->coeffs[i + 7] & 0x07);
	  b_idx++;
	  }
	}

제 4 장 AVR8bit 구현 기술 Tip 및 가이드

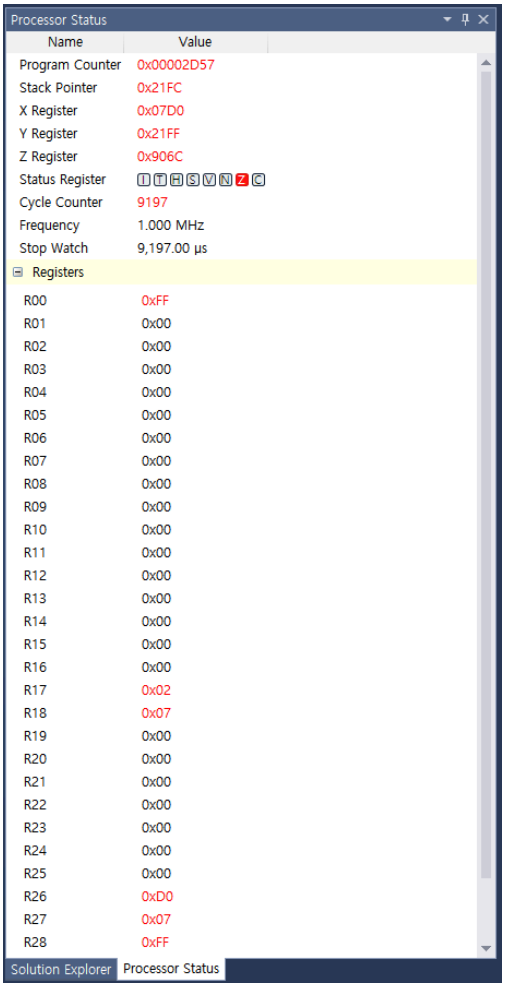
제 4.1 절 속도 및 메모리 측정

■ 속도(Clock) 측정

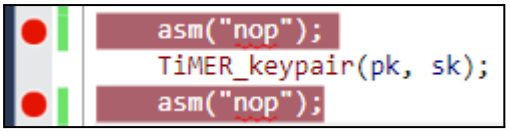
☐ Project - (프로젝트명) Properties - tool - Simulator 설정

☐ Debug - Windows - Processor Status 설정

☐ 위 두가지 설정을 진행한후 Break Point를 설정한뒤 Debugging을 시작하면 IDE 우측에 Processor Status 창이 생긴다.



☐ Processor Status의 Cycle Counter를 함수 실행시간으로 볼수있다.



☐ 위와 같이 함수의 시작과 끝에 Break Point를 걸고 디버깅을 하면 함수의 실행시간을 파악하여 성능을 측정 할 수 있다.

■ 메모리(Stack) 측정

☐ 속도 측정과 동일하게 Debugging을 통해 Processor Status에서 Stack Pointer을 이용하여 스택 사용량을 파악 할 수 있다.

제 4.2 절 malloc

■ 임베디드 환경에서의 동적 할

☐ 임베디드 환경에서는 일반적으로 동적 할당을 사용하지 않고. 전부 정적인 배열을 스택에 저장하여 사용한다.

☐ 동적 할당을 정적 배열로 바꾸기 위해서는 먼저 해당 배열이 항상 고정된 값을 할당하는지. 아니면 할당할 때마다 크기가 바뀌는지 알아야 할 필요가 있다. 만약 항상 고정된 값이 할당된다면 해당 크기만큼 정적 배열로 바꾸면 된다. 그러나 할당할 때마다 할당되는 크기가 바뀐다면 할당되는 크기의 최댓값을 알아야 한다. 해당 최댓값으로 배열을 크기를 정적 할당하면 기존에 동적 할당한 크기보다는 메모리 사용량이 증가하게 되지만. 동적 할당을 지양하는 임베디드 환경에서는 필요한 과정이다.

☐ SMAUG-T의 경우. sppoly 구조체의 sx 변수를 포인터로 두어 동적할당을 수행한다. 해당 변수는 동적 할당 할 때 SMAUG-T1 기준 최대 97바이트를 할당하며. 호출될 때마다 할당하는 크기가 다르다. 따라서 sx 변수의 최대 할당량인 97바이트만큼 할당하도록 정적으로 수정하였으며. sx 변수에 동적 할당 하는 함수가 호출되는 부분을 삭제함으로써 정적 배열로 변환 가능하다. sx 변수의 변경된 모습과 동적 할당 함수가 삭제된 것은 아래 그림에서 확인할 수 있다.

```
typedef struct {
    uint8_t* sx;
    uint8_t neg_start;
    uint8_t cnt;
} sppoly; // sparse poly
```

```
typedef struct {
    uint8_t sx[97];
    uint8_t neg_start;
    uint8_t cnt;
} sppoly; // sparse poly
```