AVR 프로그래밍

3강

정보컴퓨터공학과 권혁동

https://youtu.be/z_6OGxR2VrM

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

캐리 플래그 활용

효율적인 로드 방법

CryptoCraft LAB

캐리 플래그 활용

- 사용할 명령어 모음

명령어	동작
ADIW	워드 단위 레지스터 즉시 덧셈
ADC	레지스터 덧셈 + 캐리
LDI	레지스터에 즉시 로드
LDD	일정 번지 이후의 값 로드

캐리 플래그 활용

- 지난 강의의 8-bit 덧셈기를 16-bit로 확장
- 최하위 주소에 1을 더해주는 프로그램

```c
extern void adder16(char *x, char *y);

int main(void)
{
    char a[2] = {0x10, 0xff};
    char b[2] = {0x00, };

    adder16(a, b);
}
```

```
adder16:

MOVW R26, R24
LD R18, X+
LD R19, X+
MOVW R30, R22
LDI R20, 1

ADD R19, R20

ST Z+, R18
ST Z+, R19

RET
```

캐리 플래그 활용

- 계산 결과 예상 값과는 다름
  - 예상 결과

    0x11

    0x00

- **ADD 명령어는 캐리가 넘어가지 않는다**

캐리 플래그 활용

- ADD 명령어를 수행하는 순간 캐리 플래그가 발생함

- 캐리 플래그를 꺼내 올 수 있는 일부 명령어가 존재
  - ADC Rd, Rr
  - ROL Rd
  - ROR Rd

캐리 플래그 활용

- ADC 명령어를 통해서 캐리 플래그의 값을 가져올 수 있음
- ADC R18, R1 = R18 + R1 + C
  - 이때, R1은 제로 레지스터 이므로 사실상 R18 + C
- **따라서 덧셈을 할 때는 ADC를 사용해야 안전**
  - 단, **최하위 주소에는 ADD**를 사용

```
MOVW R26, R24
LD R18, X+
LD R19, X+
MOVW R30, R22
LDI R20, 1

ADD R19, R20
ADC R18, R1

ST Z+, R18
ST Z+, R19

RET
```

a	0x10f8
[0]	0x10
[1]	0xff
b	0x10fa
[0]	0x11
[1]	0x00

효율적인 로드 방법

- 어셈블리 코드를 조금 변경
- 배열 값 두 개를 더해서 반환하는 함수
- 16-bit 단위로 동작

```
#include <avr/io.h>

extern void adder16(char *x, char *y);


int main(void)
{
   char a[2] = {0x10, 0xff};
   char b[2] = {0x00, };

   adder16(a, b);
}
```

```
MOVW R26, R24
LD R18, X
ADIW R26, 1
LD R19, X
MOVW R30, R20

ADD R18, R19

ST Z, R18

RET
```

- 배열 형태의 값은 메모리 상에서 연속적으로 배치
  - char형 자료는 주소 값이 1차이

효율적인 로드 방법

- 어셈블리 코드를 작성

코드 분석

1. R24, R25로 넘어온 포인터 매개변수를 R26, R27(X)로 이동
2. X를 통해 첫번째 변수를(a[0]) R18에 저장
3. R26의 주소 수준을 한단계 올림
4. X를 통해 두번째 변수를(a[1]) R19에 저장
   - 디버깅에서 정상적으로 불러온 것을 확인 가능

```
MOVW R26, R24
LD R18, X
ADIW R26, 1
LD R19, X
MOVW R30, R20

ADD R18, R19

ST Z, R18

RET
```

R18	0x10
R19	0xFF
R20	0x00
R21	0x00
R22	0xFA
R23	0x10
R24	0xF8
R25	0x10
R26	0xF9
R27	0x10

효율적인 로드 방법

- 직접 주소 값을 수정하며 바꾸는 것은 매우 번거로움
- **또한 자료형에 따라서 정상적인 값 호출에 실패할 수 있음**
- 이를 위해 Post Increment / Pre Decrement가 존재
- LD 뿐만 아니라 ST에도 동일하게 적용 가능

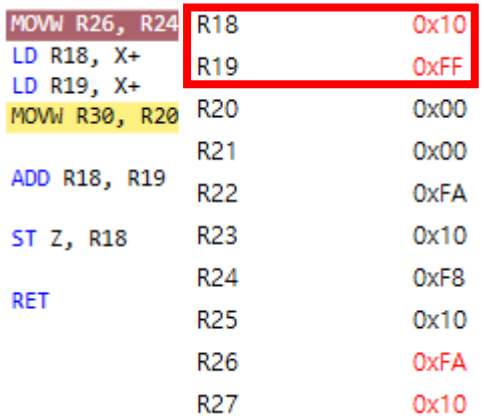
- 사용법

LD Rd, X+ ST X+, Rr

LD Rd, -X ST -X, Rr

효율적인 로드 방법

- **더 짧은 코드로 구현**된 것을 확인 가능
- 코드가 짧으므로, **동작 시간도 빨라짐**
  - 기존: 7사이클
  - 현재: 5사이클



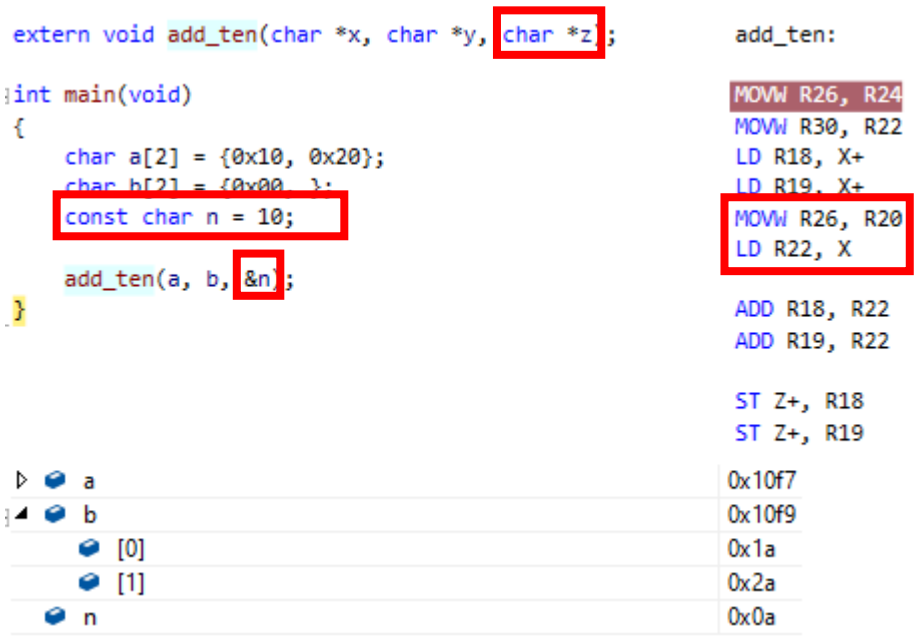
효율적인 로드 방법

- ST에도 동일하게 적용이 가능

```
MOVW R26, R24
LD R18, X+
LD R19, X+
MOVW R30, R22

ADD R18, R19

ST Z+, R18
ST Z+, R19

RET
```

효율적인 로드 방법

- 상수를 호출하는 방법
  - 숫자 10을 더해주는 프로그램 작성
- n은 상수 10을 저장하는 변수
- 상수는 바뀌지 않으므로 프로그램 내부에 하드코딩 형식으로 작성하는 것이 편리
- **매개변수를 넘기고 로드하는 시간 단축**



효율적인 로드 방법

- LDI 명령어는 정해진 **상수 값을 바로 레지스터로 로드**
- 사용법

  LD Rd, k
- 0 ~ 255(0x00 ~ 0xff)까지 로드 가능
- **단, 대상 레지스터가 R18 이상일 때만 가능**
  - R0~R17은 LDI 명령어 사용 불가

```
extern void add_ten(char *x, char *y);

int main(void)
{
    char a[2] = {0x10, 0x20};
    char b[2] = {0x00, };

    add_ten(a, b);
}
```

```
add_ten:

MOVW R26, R24
MOVW R30, R22
LD R18, X+
LD R19, X+
LDI R22, 10

ADD R18, R22
ADD R19, R22

ST Z+, R18
ST Z+, R19
```

a	0x10f8
b	0x10fa
[0]	0x1a
[1]	0x2a

효율적인 로드 방법

- 배열 인덱싱
  - 배열 첫 번째와 열 번째 값을 더하는 프로그램 작성
- 첫 번째 값 로드는 쉽지만, 열 번째 값 로드는 복잡함
  - ADIW를 통해 인덱스 뛰어넘기: 직접 주소를 계산해야 하므로 틀릴 수 있음
  - LD를 열 번해서 원하는 값을 불러오기: 동작 시간이 늘어남
- **a[9]와 같은 형식이 필요함**

```
extern void add_first_and_tenth(char *x, char *y);

int main(void)
{
    char a[10] = {0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xa0};
    char b = 0x00;

    add_first_and_tenth(a, &b);
}
```

```
add_first_and_tenth:
MOVW R30, R24
MOVW R26, R22
LD R18, Z
ADIW R30, 9
LD R19, Z

ADD R18, R19

ST X, R18

RET
```

a	0x10f1
b	0xb0

효율적인 로드 방법

- LDD는 포인터 레지스터에 저장된 값에서 **일정 번지 만큼 떨어진 값을 로드**하는데 사용
- 비슷한 명령어로 STD가 존재, 저장할 때 사용
- 사용법

  LD Rd, Z+q
- **단, X 포인터 레지스터는 LDD, STD 사용 불가**

```
extern void add_first_and_tenth(char *x, char *y);

int main(void)
{
    char a[10] = {0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xa0};
    char b = 0x00;

    add_first_and_tenth(a, &b);
}
```

```
add_first_and_tenth:
MOVW R30, R24
MOVW R26, R22
LDD R18, Z+0
LDD R19, Z+9
ADD R18, R19
ST X, R18
RET
```

a	0x10f1
b	0xb0

Q & A