AVR 어셈블리 실습

https://youtu.be/_7ZGruzDjsI

AVR

- Atmel에서 개발된 8bit RISC 마이크로 컨트롤러
  - RISC(Reduced Instruction Set Computer) : CPU 명령어의 개수를 줄여 HW 구조 단순히 만드는 방식

- 변형된 Harvard 구조
  - 프로그램과 데이터 메모리가 분리된 형태
  - 특수 명령어로 프로그램 데이터를 데이터 영역으로 읽는 것이 가능

- 6가지 계열 존재 (ATtiny, ATmega, ATxmega, Application-specific AVR, FPSLIC, 32-bit AVRs)
  - Atmega 시리즈의 **Atmega 128**이 가장 흔하게 사용
    - 133개의 RISC 명령어 사용
    - 32개의 8-bit 레지스터

AVR

- 어셈블리
  - 명령어 하나에 1대1 기계어 대응
  - 가장 효율적인 구현 가능
    - 실행 속도
    - 프로그램 Code 길이
  - 사람이 프로세서의 명령을 이해하기 쉬움

레지스터 구조 특징

- 범용 레지스터

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

  - R1 : 종료 시 항상 0으로 유지
  - R2~R17, R28,R29 : Callee Saved, 사용하기 전의 값 보존 필요
  - R26, R27 : X pointer
  - R28, R29 : Y pointer
  - R30, R31 : Z pointer

레지스터 구조 특징

- 범용 레지스터

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

- 첫번째 매개변수 주소 저장된 위치 : R24, R25
- 두번째 매개변수 주소 저장된 위치 : R22, R23
- 세번째 매개변수 주소 저장된 위치 : R20, R21
  - 위와 같은 순서로 매개변수 입력
  - 변수의 주소 값은 16bit 형태로 저장
  - 각각의 레지스터는 8bit, 주소 값은 16bit이기 때문에 위와 같이 연속된 16bit가 한 묶음

기초 어셈블리 실습

```c
extern void adder (u16 *a, u16 *b, u16 *c);

//extern void adder_32bit (u32 *a, u32 *b, u32 *c);

int main(void)
{
    u16 a[] = {0x11,0x22};
    u16 b[] = {0x55,0x66};
    u16 c[2] = {0, };

    adder(a, b, c);
```

main.c

```
.global adder
.type adder, @function

adder:

   MOVW R26, R24
   LD R18, X+
   LD R19, X+

   MOVW R26, R22
   MOVW R30, R20

   LD R20, X+
   LD R21, X+

   ADD R18, R20
   ADD R19, R21

   ST Z+, R18
   ST Z+, R19

   RET
```

adder.s

.global <symbol> : 심볼을 외부 참조 가능하게 함
.type <symbol>, @[type] : 심볼에 대한 type정의

기초 어셈블리 실습

```
.global adder
.type adder, @function
```

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
		24	25	22	23					24	25			20	21
										22	23				

MOVW Rd[도착지], Rr[출발지]

```
adder:
    MOVW R26, R24
    LD R18, X+
    LD R19, X+

    MOVW R26, R22
    MOVW R30, R20

    LD R20, X+
    LD R21, X+

    ADD R18, R20
    ADD R19, R21

    ST Z+, R18
    ST Z+, R19

    RET
```

LOAD Rd[도착지], X[포인터 reg]

ADD Rd[도착지], Rr[출발지]

ST Z[포인터 reg], Rr[출발지]

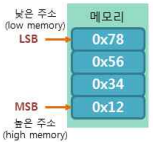
```
int main(void)
{
    u16 a[] = {0x11,0x22};
```

리틀엔디안 방식
0x11이 아닌 **0x22가 R18에 LOAD**

리틀엔디안 방식

0x12345678

Byte 단위 저장 순서
: 0x78563412



MOVW	WORD단위로 이동(복사)
LD	메모리에서 레지스터로 LOAD
ADD	레지스터 덧셈
ST	레지스터에서 메모리로 STORE
RET	함수 반환

기초 어셈블리 실습

```
.global adder
.type adder, @function
```

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
		24	25	22	23					24 22	25 23			20	21

```
adder:
    MOVW R26, R24
    LD R18, X+
    LD R19, X+

    MOVW R26, R22
    MOVW R30, R20

    LD R20, X+
    LD R21, X+

    ADD R18, R20
    ADD R19, R21

    ST Z+, R18
    ST Z+, R19

    RET
```

```
int main(void)
{
    u16 a[] = {0x11,0x22};
    u16 b[] = {0x55,0x66};
    u16 c[2] = {0, };

    adder(a, b, c);
```

MOVW	WORD단위로 이동(복사)
LD	메모리에서 레지스터로 LOAD
ADD	레지스터 덧셈
ST	레지스터에서 메모리로 STORE
RET	함수 반환

기초 어셈블리 실습

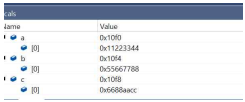
```
extern void adder_32bit (u32 *a, u32 *b, u32 *c);

int main(void)
{
    u32 a[] = {0x11223344};
    u32 b[] = {0x55667788};
    u32 c[1] = {0, };

    //adder(a, b, c);
    adder_32bit(a, b, c);
```

R18	0x44
R19	0x33
R20	0x22
R21	0x11

R22	0x88
R23	0x10
R24	0xF0
R25	0x10
R26	0xF5

R22	0x88
R23	0x77
R24	0x66
R25	0x55
R26	0xF8

```
adder_32bit:

    MOVW R26, R24
    MOVW R30, R20

    LD R18, X+
    LD R19, X+
    LD R20, X+
    LD R21, X+

    MOVW R26, R22

    LD R22, X+
    LD R23, X+
    LD R24, X+
    LD R25, X+


    ADD R18, R22
    ADD R19, R23
    ADD R20, R24
    ADD R21, R25

    ST Z+, R18
    ST Z+, R19
    ST Z+, R20
    ST Z+, R21

    RET
```



기초 어셈블리 실습

- 로테이션 : 비트 이동 후, 끝 자리에 반대쪽 값으로 대체됨
- 시프트 : 비트 이동 후, 끝 자리에 0으로 채움

ROL	**왼쪽**으로 로테이션
ROR	**오른쪽**으로 로테이션
LSL	**왼쪽**으로 시프트
LSR	**오른쪽**으로 시프트
BLD	레지스터의 특정 비트로 이동
BST	플래그를 레지스터의 특정 비트로 이동

기초 어셈블리 실습

```
shift:

    MOVW R26, R24
    LD R18, X+
    LD R19, X+

    LSL R18
    ROL R19
    ADC R18, R1

    ST X+, R18
    ST X+, R19

    RET
```

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

typedef uint16_t u16;

extern void shift(u16 *a, u16 *b);

int main(void)
{
    u16 a[] ={0x8081};
    u16 b[1]={0,};

    shift(a,b);
}
```

0X8081 → 1000 0000 1000 0001 왼쪽 로테이션

								C	캐리 플래그
ROL R18	0	0	0	0	0	0	1	1	0	
ROL R19	0	0	0	0	0	0	0	1	0	원하는 결과값
ROL R18	0	0	0	0	0	0	1	0	1	
ROL R19	0	0	0	0	0	0	0	1	0	결과값

기초 어셈블리 실습

```
shift:
    MOVW R26, R24
    LD R18, X+
    LD R19, X+

    LSL R18
    ROL R19
    ADC R18, R1

    ST X+, R18
    ST X+, R19

    RET
```

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

typedef uint16_t u16;

extern void shift(u16 *a, u16 *b);

int main(void)
{
    u16 a[] ={0x8081};
    u16 b[1]={0,};

    shift(a,b);
}
```

0X8081 → 1000 0000 1000 0001 왼쪽 로테이션

C 캐리 플래그

LSL R18	0	0	0	0	0	0	1	0	1
ROL R19	0	0	0	0	0	0	0	1	1
ADC R18, R1	0	0	0	0	0	0	1	1	0

기초 어셈블리 실습

```
shift:
    MOVW R26, R24
    LD R18, X+
    LD R19, X+

    BST R18,0
    LSR R19
    ROR R18
    BLD R19, 7
```

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

typedef uint16_t u16;

extern void shift(u16 *a, u16 *b);

int main(void)
{
    u16 a[] ={0x8081};
    u16 b[1]={0,};

    shift(a,b);
}
```

0X8081 → 1000 0000 1000 0001 에 대한 오른쪽 로테이션

ROR R19

								C	T	
1	1	0	0	0	0	0	0	0	0	원하는 결과값
0	1	0	0	0	0	0	0	1	0	결과값
1	1	0	0	0	0	0	0	0	0	

비트 복사 플래그

우연히 발생한 결과로, 옳은 구현 X

기초 어셈블리 실습

```
shift:
    MOVW R26, R24
    LD R18, X+
    LD R19, X+

    BST R18,0
    LSR R19
    ROR R18
    BLD R19, 7
```

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

typedef uint16_t u16;

extern void shift(u16 *a, u16 *b);

int main(void)
{
    u16 a[] ={0x8081};
    u16 b[1]={0,};

    shift(a,b);
}
```

0X8081 → 1000 0000 1000 0001 에 대한 로테이션

									C	T
BST R18, 0	1	0	0	0	0	0	0	1	0	1
LSR R19	0	1	0	0	0	0	0	0	0	1
ROR R18	0	1	0	0	0	0	0	0	1	1
BLD R19, 7	1	1	0	0	0	0	0	0	1	0

비트 복사 플래그

Q & A