Homework 03: Assembly Language

Tae-Hyun Oh
Associate Professor
Dept. Electrical Engineering
POSTECH, Korea

Slides by
Youngjoo Lee

EPIC LAB

Intro to Assembly Language

Intro to Assembly Language

Inline assembly 작성법

- ✓ 전체 틀은 기존의 C와 유사하며, Assembly로 작성하는 부분을 asm();으로 묶음.
- ✓ 각 명령어의 끝에 \n\t를 붙여 명령어 구분함.
- ✓ asm();에 있는 명령에 의해 c변수들의 입력/출력이 생기는 경우 코드 뒤에 input & output operands 작성함.

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

int main () {

    int i, sum, N;

    N = 2000000;
    sum = 0;

    int *data = (int*)malloc(sizeof(int)*N);

    for (i = 0; i < N; i++)
        data[i] = i+1;

    printf("Before: %d\n", sum);

    asm(
        "mov r4, #0\n\t"                // reg for address offset
        "mov r5, #0\n\t"                // reg_sum = 0
        "mov r6, #0\n\t"                // i = 0
        "ldr r8, %[N]\n\t"              // reg_N = N
        "Loop:\n\t"
        "ldr r7, [%[data], r4]\n\t"     // reg_data = data[i]
        "add r5, r5, r7\n\t"            // reg_sum += reg_data
        "add r4, r4, #4\n\t"            // offset += 4
        "add r6, r6, #1\n\t"            // i++
        "cmp r6, r8\n\t"                // check i < reg_N
        "blt Loop\n\t"

        "str r5, %[sum]\n\t"            // sum = reg_sum

        :
        // input operands
        :
        [N] "m"(N), [data] "r"(data), [sum] "m"(sum)
        // output operands
        // "r" : A register operand is allowed provided that it is
        //       in a general register.
        // "m" : A memory operand is allowed, with any kind of address
        //       that the machine supports in general.
        :
        "r4", "r5", "r6", "r7"
        // clobbers
        // : The clobber list informs the compiler that the assembly
        //   code modifies these registers.
    );

    printf("After: %d\n", sum);

    return 0;
}
```

Intro to Assembly Language

Inline assembly 작성법

- ✓ Input & output operands
  - [SymbolName] constraint (C-expression)
  - Constraint
    - “r” : (c-expression)의 값을 symbolname에 저장
    - “m” : (c-expression)의 주소를 symbolname에 저장
    - 사용할 때 %[SymbolName]을 register 대신 적음.
  - Clobbers
    - Inline assembly에서 수정되는 register의 목록

```
asm(
    "mov r4, #0\n\t"                // reg for address offset
    "mov r5, #0\n\t"                // reg_sum = 0
    "mov r6, #0\n\t"                // i = 0
    "ldr r8, %[N]\n\t"              // reg_N = N
    "Loop:\n\t"
    "ldr r7, [%[data], r4]\n\t"     // reg_data = data[i]
    "add r5, r5, r7\n\t"            // reg_sum += reg_data
    "add r4, r4, #4\n\t"            // offset += 4
    "add r6, r6, #1\n\t"            // i++
    "cmp r6, r8\n\t"                // check i < reg_N
    "blt Loop\n\t"

    "str r5, %[sum]\n\t"            // sum = reg_sum

    :
    // input operands
    :
    [N] "m"(N), [data] "r"(data), [sum] "m"(sum)
    // output operands
    // "r" : A register operand is allowed provided that it is
    //       in a general register.
    // "m" : A memory operand is allowed, with any kind of address
    //       that the machine supports in general.
    :
    "r4", "r5", "r6", "r7"
    // clobbers
    // : The clobber list informs the compiler that the assembly
    //   code modifies these registers.
);
```

Intro to Assembly Language

Instruction set

Instruction	operation
add r1, r2, r3	r1 ← r2 + r3
mov r1, #0	r1 ← 0
ldr r1, [r2,r3]	r1 ← mem[r2 + r3]
str r1, [r2,r3]	mem[r2 + r3] ← r1
cmp r1, r2	r1 - r2 의 결과를 status register에 저장
B{cond} label	If cond : goto label

```
asm(
    "mov r4, #0\n\t"                // reg for address offset
    "mov r5, #0\n\t"                // reg_sum = 0
    "mov r6, #0\n\t"                // i = 0
    "ldr r8, %[N]\n\t"              // reg_N = N
    "Loop:\n\t"
    "ldr r7, [%[data], r4]\n\t"     // reg_data = data[i]
    "add r5, r5, r7\n\t"            // reg_sum += reg_data
    "add r4, r4, #4\n\t"            // offset += 4
    "add r6, r6, #1\n\t"            // i++
    "cmp r6, r8\n\t"                // check i < reg_N
    "blt Loop\n\t"

    "str r5, %[sum]\n\t"            // sum = reg_sum

    :
    // input operands
    :
    [N] "m"(N), [data] "r"(data), [sum] "m"(sum)
    // output operands
    // "r" : A register operand is allowed provided that it is
    //       in a general register.
    // "m" : A memory operand is allowed, with any kind of address
    //       that the machine supports in general.
    :
    "r4", "r5", "r6", "r7"
    // clobbers
    // : The clobber list informs the compiler that the assembly
    //   code modifies these registers.
);
```

- ✓ B{cond}
  - Cond는 lt(less than), gt(greater than), eq(equal), le(less than or equal), ge(greater than or equal) 이 있음.
  - 이전 연산에 의해 status register에 저장된 값을 참조
- ✓ Label
  - 우측의 예시에서 Loop:\n\t 가 이동할 주소를 지시하는 label

Intro to Assembly Language

Example

✓ 1부터 N까지 더해서 sum에 저장

C

```c
#include <stdlib.h>

int main ()
{
    int i,sum,N;
    N=2000000;
    int *data=(int*)malloc(sizeof(int)*N);
    for(i=0;i<N;i++) {
        data[i]=(i+1);
    }
    sum=0;
    for(i=0;i<N;i++) {
        sum=sum+data[i];
    }
}
```

Inline assembly

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

int main () {

    int i, sum, N;

    N = 2000000;
    sum = 0;

    int *data = (int*)malloc(sizeof(int)*N);

    for (i = 0; i < N; i++)
        data[i] = i+1;

    printf("Before: %d\n", sum);

    asm(
        "mov r4, #0\n\t"                // reg for address offset
        "mov r5, #0\n\t"                // reg_sum = 0
        "mov r6, #0\n\t"                // i = 0
        "ldr r8, %[N]\n\t"              // reg_N = N
        "Loop:\n\t"
        "ldr r7, [%[data], r4]\n\t"     // reg_data = data[i]
        "add r5, r5, r7\n\t"            // reg_sum += reg_data
        "add r4, r4, #4\n\t"            // offset += 4
        "add r6, r6, #1\n\t"            // i++
        "cmp r6, r8\n\t"                // check i < reg_N
        "blt Loop\n\t"

        "str r5, %[sum]\n\t"            // sum = reg_sum

        :
        // input operands
        :
        [N] "m"(N), [data] "r"(data), [sum] "m"(sum)
        // output operands
        // "r" : A register operand is allowed provided that it is
        //       in a general register.
        // "m" : A memory operand is allowed, with any kind of address
        //       that the machine supports in general.
        :
        "r4", "r5", "r6", "r7"
        // clobbers
        // : The clobber list informs the compiler that the assembly
        //   code modifies these registers.
    );

    printf("After: %d\n", sum);

    return 0;
}
```

Intro to Assembly Language

Example

✓ C언어로 작성한 코드를 –S 옵션을 주어 컴파일하면 assembly language로 변환된 파일을 확인할 수 있음.

- **`$ gcc –S <filename>.c –o <filename>.s`**
- **`$ vi <filename>.s`**

*예시 사진은 x86 instruction 기반 assembly이므로 라즈베리파이의 결과와 다름.

```c
int main () {

    int i, sum, N;

    N = 2000000;
    sum = 0;

    int *data = (int*)malloc(sizeof(int)*N);

    for (i = 0; i < N; i++)
        data[i] = i+1;

    printf("Before: %d\n", sum);

    for (i = 0; i < N; i++)
        sum += data[i];
```

```
main:
.LFB2:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl    $2000000, -12(%rbp)
    movl    $0, -8(%rbp)
    movl    -12(%rbp), %eax
    cltq
    salq    $2, %rax
    movq    %rax, %rdi
    call    malloc
    movq    %rax, -24(%rbp)
    movl    $0, -4(%rbp)
    jmp .L2
.L3:
    movl    -4(%rbp), %eax
    cltq
    leaq    0(,%rax,4), %rdx
    movq    -24(%rbp), %rax
    addq    %rdx, %rax
    movl    -4(%rbp), %edx
    addl    $1, %edx
    movl    %edx, (%rax)
    addl    $1, -4(%rbp)
.L2:
    movl    -4(%rbp), %eax
    cmpl    -12(%rbp), %eax
    jl  .L3
    movl    -8(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, -4(%rbp)
    jmp .L4
```

Intro to Assembly Language

Example

- ✓ <filename>.s assembly 파일을 참고하여 inline assembly를 직접 작성
- ✓ Instruction set은 아래 링크를 참고

- https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set

Example Result

✓ Running Time 출력에 대해서는 아래 예시와 같이 time 기능을 사용해도 무관함

```
pi@raspberrypi:~/Desktop $ gcc add_c.c
pi@raspberrypi:~/Desktop $ time ./a.out

real    0m0.148s
user    0m0.138s
sys     0m0.010s
pi@raspberrypi:~/Desktop $ gcc add_asm.c
pi@raspberrypi:~/Desktop $ time ./a.out

real    0m0.091s
user    0m0.041s
sys     0m0.050s
```

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel,* as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

Problem Definition

Problem Definition

컴퓨터 프로그래밍 언어에 따른 연산처리속도 비교

(* 라즈베리파이에서 진행)

1. 실행 예시: input argument 사용 (argc, **argv)
   1. `$ ./insertion.out N`
   2. `$ ./merge.out N`
2. 양의 정수 N에 대해, 1 ~ N까지의 정수를 random한 순서로 중복없이 생성
3. Insertion Sort를 C와 inline assembly로 각각 작성 및 개별 함수로 정의
4. Merge Sort를 C와 inline assembly로 각각 작성 및 개별 함수로 정의
5. Sorting 부분만 inline assembly로 구현 (그 외 부분은 C로 작성)
6. N ≤ 20 일 때, C와 inline assembly로 구현한 함수 각각의 sorting 전/후를 출력
7. 동일한 입력에 대해 각 함수의 Execution Time을 출력

✓ **정렬 기준은 오름차순**

Problem Definition

Source code structure example

✓ 함수 call, return type, parameter 등… 세부 구현 사항은 자유.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// user defined function
void insertion_C(<inputs>);
void insertion_ASM(<inputs>);

int main(int argc, char* argv[]) {
    srand(time(NULL));

    // user input
    // variable initialization

    // print data before sorting

    // measuring time, recommended to use clock_gettime()
    insertion_C(<inputs>);
    // measuring time

    // measuring time
    insertion_ASM(<inputs>);
    // measuring time

    // print data after sorting
    // print run time

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// user defined function
void mergesort_C(<inputs>);
void merge_C(<inputs>);
void mergesort_ASM(<inputs>);
void merge_ASM(<inputs>);

int main(int argc, char* argv[]) {
    srand(time(NULL));

    // user input
    // variable initialization

    // print data before sorting

    // measuring time, recommended to use clock_gettime()
    mergesort_C(<inputs>);
    // measuring time

    // measuring time
    mergesort_ASM(<inputs>);
    // measuring time

    // print data after sorting
    // print run time

    return 0;
}
```

✓ Input / Output format example

```
kwonsh01@raspberrypi:~/Desktop/ASM $ ./insertion.out 10
Before sort     : [ 8 2 1 5 6 10 9 3 7 4 ]
After sort   (C): [ 1 2 3 4 5 6 7 8 9 10 ]
After sort (ASM): [ 1 2 3 4 5 6 7 8 9 10 ]
Execution Time   (C): 0.000004[s]
Execution Time (ASM): 0.000001[s]
```

```
kwonsh01@raspberrypi:~/Desktop/ASM $ ./merge.out 10
Before sort     : [ 8 10 9 7 4 3 5 1 2 6 ]
After sort   (C): [ 1 2 3 4 5 6 7 8 9 10 ]
After sort (ASM): [ 1 2 3 4 5 6 7 8 9 10 ]
Execution Time   (C): 0.000008[s]
Execution Time (ASM): 0.000005[s]
```

Problem Definition

Merge sort 구현 시 참고 사항

✓ 기본적으로 recursion 부분을 inline assembly로 구현하지 않아도 됨.

- ✓ Sorting과 관련된 연산 및 algorithm 부분만(conquer part) 구현하는 것을 기본으로 함
- ✓ Code structure example.(algorithm만 지킨다면 structure 및 function의 모든 것은 자유.)

```
void mergesort(<inputs>) {
    // local variable

    // recursion
    mergesort(<inputs>);
    mergesort(<inputs>);

    asm(
        // inline assembly
    );

    return;
}
```

or

```
void mergesort(<inputs>) {
    // local variable

    // recursion
    mergesort(<inputs>);
    mergesort(<inputs>);

    // function call
    merge(<inputs>);

    return;
}

void merge(<inputs>) {
    // local variable

    asm(
        // inline assembly
    );
    return;
}
```

or

Recursion with inline assembly

```
void mergesort(<inputs>) {
    // local variable

    asm(
        // inline assembly
    );

    return;
}
```

etc.

✓ **Recursion 구현 시, function call과 return from subroutine 부분을 모두 inline assembly로 구현한 경우에는 extra credit 30점을 부여함.**

Problem Definition

Merge sort 구현 시 참고 사항

- ✓ **Inline assembly로 recursion 동작 구현 시 해당 내용 또한 리포트에 포함할 것.**

그 외

- ✓ N의 최댓값은 따로 정해져 있지 않음. 하지만 N 값을 증가시키면서 실험을 한 내용이 리포트에 포함되어야 함.
- ✓ 출력 양식 준수.

For N ≤ 20

```
Before^sort^^^^^:^[^4^3^2^1^]\n
After^sort^^^^(C):^[^1^2^3^4^]\n
After^sort^(ASM):^[^1^2^3^4^]\n
Execution^Time^^^^(C):^0.000000[s]\n
Execution^Time^(ASM):^0.000000[s]\n
```

For N > 20

```
Execution^Time^^^^(C):^0.000000[s]\n
Execution^Time^(ASM):^0.000000[s]\n
```

Submission & Evaluation

Submission

조교가 검사할 수 있는 source code와 결과보고서 PDF를 제출

- ✓ Due date: 4/25 (Thu) 23:59
- ✓ 제출 방식: 리포트와 각 소스코드 파일을 **학번_이름.zip**으로 plms에 제출
  - **학번_insertion.c, 학번_merge.c, 학번_report.pdf**
- ✓ 과제에 관한 질문은 Q&A 게시판 활용
- ✓ 문법에 관련된 질문은 링크 및 강의자료 참고
- ✓ 기재된 양식 외 제출시 페널티 부여

- ✓ 추가적인 gcc 옵션 없이 라즈베리파이 환경에서 채점 예정.
  - `$ gcc -o merge.out 학번_merge.c`
  - `$ gcc -o insertion.out 학번_insertion.c`

- ✓ 담당조교 - 권순현 (soonhyun.kwon@postech.ac.kr)

Submission

다음 내용을 포함하여 결과보고서 작성

1. 각 알고리즘에 대한 간단한 설명
2. C 및 assembly 코드가 정상적으로 동작하고 있음을 보임
3. C 및 assembly 코드의 N 값에 따른 수행 시간 비교분석 및 고찰
4. Sorting algorithm 비교분석 및 고찰

* Source code에 대해서 line-by-line으로 설명할 필요 X

Evaluation

Source code (70)

1. Insertion sorting algorithm implementation in C & Assembly (30)
2. Merge sorting algorithm implementation in C & Assembly (30)
3. Input / Output format (10)

Report (30)

1. Preliminaries, background description (10)
2. Comparative analysis between languages & sorting algorithms (10)
3. Discussion (10)

Extra credit (30)

1. Recursion implementation of merge sorting in inline assembly (30)

0 credits for {cheating, late submission}