어셈블리 실습 2

Tags

1. 실습 내용

> 💡 실습에 쓰이는 c테스트 파일들은 /mnt/SysSW/Prac2 아래에 있어서, 실습 진행하는 디렉토리에 복사해서 사용하시면 됩니다.

A. 실습 목표

- **반복문의 변환 방식을 이해하고, 반복문이 들어간 함수를 구현해보자**
- GCC 최적화 옵션(`Og`, `O1`)을 활용해서 최적화 단계에서 변환 차이
  - `-Og` (Optimize for Debugging)
    - 디버깅에 필요한 정보를 최대한 유지하면서, -O1보다 적은 최적화를 수행
  - `-O1` (Optimize)
    - **일반적인 최적화:** `O1`은 컴파일 시간과 실행 속도 사이의 균형을 맞추는 기본적인 최적화 수준
- 테스트 코드

```
// loop_do_while_sum_ctest.c
#include <stdio.h>

// do-while 루프: 배열의 모든 요소를 더하기
int do_while_loop_sum(int* arr, int n) {
    int i = 0;
    int sum = 0;
    do {
        sum += arr[i];
        i++;
    } while (i < n);
    return sum;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    // 각 루프에서 계산된 합을 저장
    int sum_do_while = do_while_loop_sum(arr, n);
     // 결과 출력 및 검증
```

```
    printf("Sum using do-while loop: %d\n", sum_do_while);

    return 0;
}
```

- **코드를 `loop_sum_ctest.c` 파일로 저장합니다.**
- **GCC 명령어 수정**

  각 최적화 옵션으로 컴파일하여 어셈블리 코드를 생성합니다.

```
gcc -S -Og loop_sum_ctest.c -o loop_sum_ctest_Og.s
```

- 생성된 어셈블리어 코드를 살펴보면?

```
do_while_loop_sum:
.LFB23:
    .cfi_startproc            ; Call Frame Information 시작 (디버깅용 스
    endbr64                   ; Indirect branch protection (Intel CET

    movl    $0, %edx          ; sum = 0 초기화 (edx 레지스터에 저장)
    movl    $0, %eax          ; i = 0 초기화 (eax 레지스터에 저장)

.L2:
    movslq  %eax, %rcx        ; i를 64비트 rcx로 확장 (32비트 -> 64비트)
    addl    (%rdi,%rcx,4), %edx ; sum += arr[i]; 배열의 i번째 요소 값을 e
    addl    $1, %eax          ; i++ (eax 증가)

    cmpl    %esi, %eax        ; i < n 검사 (esi에는 배열의 길이 n이 저장됨)
    jl      .L2               ; i < n이면 .L2로 점프 (루프 반복)

    movl    %edx, %eax        ; 최종 합계를 eax에 저장
    ret                       ; 함수 반환 (eax에 최종 합계가 저장됨)

    .cfi_endproc              ; Call Frame Information 종료
```

- **`movslq %eax, %rcx` ?**
  - **mov**e **s**igned **l**ong to **q**uad
    - **sl**: "signed long"의 약자로, 32비트 값 ⇒ **q**: "quad"의 약자로, 64비트 값
    - **부호 확장**(Sign-extension)을 수행
      - [리마인드] `%eax`의 최상위 비트가 1이면 `%rcx`의 상위 32비트를 1로 채우고, 0이면 상위 32비트를 0으로 채움
  - 왜? 64비트 머신이기 때문에, 64비트 주소 계산이 필요함
    - 이에 따라 확장이 필요함

- `addl (%rdi,%rcx,4), %edx`에서 `4`의 의미?
  - `addl (%rdi,%rcx,4), %edx` 명령어는 `%rdi`가 가리키는 메모리 위치에서 `%rcx` 인덱스를 사용하여 배열의 특정 요소를 읽어와 `%edx`에 **더하는 작업을 수행**
  - 여기서 `4`**는 배열 요소가 4바이트 크기**임을 나타냄

B. 요걸 NASM을 위해서 intel 문법 방식으로 표현하면 ?

Step 1: `Intel 문법용` 어셈블리 코드 컴파일

- `인텔 문법`은 `mov eax, edx`처럼 목적지-출발지 순서이며, AT&T 문법은 `movl %edx, %eax`처럼 출발지-목적지 순서로 배치

- 목적어(operand)의 데이터 흐름이 반대임을 항상 염두하자

- 쉽게 `%` 기호가 있는지 없는지 여부로 인텔 문법인지 AT&T 문법인지는 알아챌 수 있음

```
section .text
global do_while_loop_sum

do_while_loop_sum:
    ; 시작
    mov edx, 0                ; sum = 0 초기화 (edx에 저장)
    mov eax, 0                ; i = 0 초기화 (eax에 저장)

.loop_start:
    movsxd rcx, eax           ; i 값을 64비트로 부호 확장하여 rcx에 저장
    add edx, dword [rdi + rcx * 4] ; sum += arr[i] (rdi 배열의 i번째 요소를
    add eax, 1                ; i++ (i를 1 증가시킴)

    cmp eax, esi              ; i < n 검사 (esi에 n이 저장됨)
    jl .loop_start            ; i < n이면 .loop_start로 점프하여 루프 반복

    mov eax, edx              ; 최종 합계를 eax에 저장
    ret                       ; 함수 반환 (eax에 최종 합계가 저장됨)
```

- `movsxd rcx, eax`
  - **명령어 의미:** `movsxd`는 "Move with Sign Extend"의 약자
  - intel 문법에서 사용되는 명령어
    - `movslq`는 AT&T 문법에서 32비트 확장용 `movsxd`

    - **intell general-purpose sign-extension instruction**
        - 즉, 8비트, 16비트, 32비트 값을 32비트 또는 64비트 값으로 부호 확장

- `add edx, dword [rdi + rcx * 4]`
    - **명령어 의미**: `add`는 더하기 연산을 수행하는 명령어
        - 여기;서 `edx`에 배열 요소를 더하는 역할을 함
    - **작동 방식**: `[rdi + rcx * 4]`는 메모리 주소를 지정하는 방식으로, `rdi` 레지스터가 배열의 시작 주소를 가리키고, `rcx`가 배열의 인덱스 역할을 함
        - `rdi + rcx * 4`는 `arr[i]`의 주소를 의미하고, 그 값을 `dword` (32비트) 크기로 가져와 `edx`에 더함

Step 2: 어셈블리 코드 컴파일

작성한 어셈블리 코드를 NASM을 사용하여 오브젝트 파일로 컴파일함

```
nasm -f elf64 do_while_loop_sum.asm -o do_while_loop_sum.o
```

- 위 명령어는 64비트 ELF 형식으로 오브젝트 파일 `do_while_loop_sum.o`를 생성

Step 3: C 파일 작성 (`loop_do_while_sum_test.c`)

C 파일에서 `do_while_loop_sum` 함수를 `extern` 키워드를 사용하여 외부 함수로 선언하고, 이를 호출하여 사용

C 코드 (`loop_do_while_sum_test.c`)

```
// loop_do_while_sum_test.c
#include <stdio.h>

extern int do_while_loop_sum(int* arr, int n);  // 외부 어셈블리 함수 선언

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    // do-while 루프를 사용하여 배열 요소의 합 계산
    int sum_do_while = do_while_loop_sum(arr, n);

    // 결과 출력 및 검증
    printf("Sum using do-while loop: %d\n", sum_do_while);

    return 0;
```

```
}
```

Step 4: C 파일과 오브젝트 파일 링크

이제 `loop_do_while_sum_test.c` 파일과 `do_while_loop_sum.o` 파일을 함께 링크하여 실행 파일을 생성

```
gcc loop_do_while_sum_test.c do_while_loop_sum.o -o loop_do_while_sum_
test
```

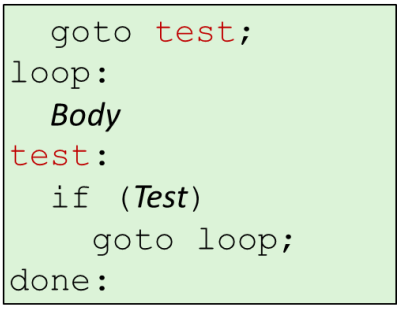
- 위 명령어는 C 파일과 NASM으로 컴파일한 오브젝트 파일을 함께 링크하여 `loop_do_while_sum_test` 실행 파일 생성.

Step 5: 실행 파일 실행

```
./loop_do_while_sum_test
```

C. 실습 내용

- 위의 NASM 코드를 이용해서 jump-to-middle 방식의 변환과 guarded_do 변환 방 의 어셈블리 코드를 작성해보자
  - 새로 명령어를 짜는 것이 아니고, 위의 NASM 코드의 명령어 재배치가 목적
- **Jump to Middle 변환 기반 코드 프레임**
  - `jump_to_middle_loop_sum.asm` 샘플 (todo: 부분을 채우시오)




```
section .text
global jump_to_middle_loop_sum

while_loop_sum:
    ; todo: sum과 i를 초기화



    jmp .loop_check           ; 처음에 조건 검사 부분으로 점프
```

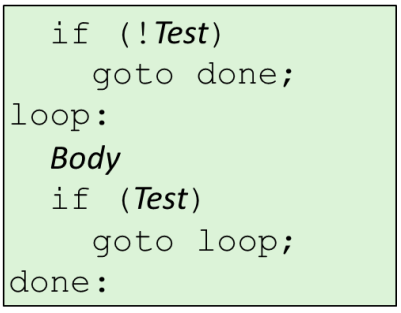
```
.loop_start:
    ; todo: body 부분 채우자


.loop_check:
    ; todo: test 부분 채우자

    jl .loop_start            ; i < n이면 .loop_start로 점프하여 루프 반복

    mov eax, edx              ; 최종 합계를 eax에 저장
    ret                       ; 함수 반환 (eax에 최종 합계가 저장됨)
```

- `guarded_do_loop_sum.asm` 샘플 (todo: 부분을 채우시오)




```
section .text
global guarded_do_loop_sum

guarded_do_loop_sum:
    ; todo: 초기화


    ; todo: 초기 조건 검사


.loop_start:
      ; todo: body


      ; todo: test

.loop_end:
    mov eax, edx              ; 최종 합계를 eax에 저장
    ret                       ; 함수 반환 (eax에 최종 합계가 저장됨)
```

- 사용하는 테스트 C 코드

```
// loop_sum_conversion_test.c
#include <stdio.h>

// NASM으로 구현된 jump-to-middle 방식의 while 루프
extern int jump_to_middle_loop_sum(int* arr, int n);

// NASM으로 구현된 guarded-do 방식의 for 루프
extern int guarded_do_loop_sum(int* arr, int n);

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    // NASM에서 구현된 각 루프 방식으로 계산된 합을 저장
    int sum_jump_to_middle = jump_to_middle_loop_sum(arr, n);
    int sum_guarded_do = guarded_do_loop_sum(arr, n);

    // 결과 출력 및 검증
    printf("Sum using jump-to-middle loop: %d\n", sum_jump_to_middle);
    printf("Sum using guarded-do loop: %d\n", sum_guarded_do);

    return 0;
}
```

D. 제출물

1. **어셈블리어 코드**
2. **테스트 C 코드 실행 화면**
3. **아래 C 코드의 `for` 문에 해당하는 어셈블리어 파일을 생성하고, 함수 부분을 찾아 주석을 완성**

```
// for_loop_ctest.c
#include <stdio.h>

// for 루프: 배열의 모든 요소를 더하기
int for_loop_sum(int* arr, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

```
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    // for 루프에서 계산된 합을 저장
    int sum_for = for_loop_sum(arr, n);

    // 결과 출력 및 검증
    printf("Sum using for loop: %d\n", sum_for);

    return 0;
}
```

4. **두 벡터의 내적(dot product)을 계산하는 어셈블리어 코드를 완성하시오.**
    - 곱하기를 위해서 사용하는 명령어
        - `imul` 명령어는 x86 및 x86-64 어셈블리어에서 **부호 있는 정수 곱셈**을 수행하는 명령어
    - 함수 인자 설명
        - `vec1`과 `vec2` 두 개의 배열을 정의하고, 배열의 길이 `n`을 계산

```
section .text
global dot_product

; int dot_product(int* vec1, int* vec2, int n)
dot_product:
    ; 초기화
    mov eax, 0              ; sum = 0 (결과값을 저장할 레지스터)
    mov ecx, 0              ; i = 0 (인덱스 레지스터)

.loop_start:
    ; i < n 검사




    ; vec1[i] * vec2[i] 계산




    ; sum += vec1[i] * vec2[i]
```

```
    ; i++
    inc ecx                 ; 인덱스 i를 증가

    jmp .loop_start         ; 루프 반복

.loop_end:
    ret                     ; 결과 반환 (eax에 최종 내적 값 저장됨)
```

5. **위의 어셈블리어 코드를 아래의 테스트 코드를 이용해서 실행하고, 결과 캡쳐**

```
// dot_product_test.c
#include <stdio.h>

// 어셈블리로 구현된 dot_product 함수 선언
extern int dot_product(int* vec1, int* vec2, int n);

int main() {
    // 테스트할 두 벡터 정의
    int vec1[] = {1, 2, 3, 4, 5};
    int vec2[] = {6, 7, 8, 9, 10};
    int n = sizeof(vec1) / sizeof(vec1[0]);  // 벡터 길이

    // dot_product 함수 호출
    int result = dot_product(vec1, vec2, n);

    // 결과 출력
    printf("Dot product of vec1 and vec2: %d\n", result);

    return 0;
}
```