어셈블리 실습 5

Tags

실습 관련 슬라이드

- **다양한 명령어들 (Lots of Instructions):**
  - 부동소수점 연산과 벡터 연산을 수행하기 위해 다양한 명령어와 형식이 사용됨
  - 예: addps, mulps, subps는 단일 정밀도(float) 벡터 연산, addpd, mulpd, subpd는 배정밀도(double) 벡터 연산.
- **비교명령어**
  - ucomiss: **float** 비교, ucomisd: **double** 비교
  - **조건 코드 설정 (Condition Codes):**
    - **CF (Carry Flag):** 값이 작은 경우 설정
    - **ZF (Zero Flag):** 값이 같은 경우 설정
    - **PF (Parity Flag):** NaN (Not a Number)인 경우 설정
- **XMM 레지스터를 0으로 초기화**하는 명령어:
  - xorpd %xmm0, %xmm0
    - **설명:** %xmm0 레지스터의 값을 XOR 연산하여 0으로 설정

Floating-point movement instructions - 예시

dongguk UNIVERSITY

```
float float_mov(float v1, float *src, float *dst) {
    float v2 = *src;
    *dst = v1;
    return v2;
}
```

```
float_mov:
  vmovaps %xmm0, %xmm1      Copy v1
  vmovss  (%rdi), %xmm0     Read v2 from src
  vmovss  %xmm1, (%rsi)     Write v1 to dst
  ret                       Return v2 in %xmm
```

여기서 v는 vector 연산을 지원한다는 의미

Instruction	Source	Destination	Description
vmovss	M_{32}	X	Move single precision
vmovss	X	M_{32}	Move single precision
vmovsd	M_{64}	X	Move double precision
vmovsd	X	M_{64}	Move double precision
vmovaps	X	X	Move aligned, packed single precision
vmovapd	X	X	Move aligned, packed double precision

Figure 3.46 **Floating-point movement instructions.** These operations transfer values between memory and registers, as well as between pairs of registers. (X: XMM register (e.g., %xmm3); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

37

문제1. Scalar 연산

C 코드에서는 배열의 원소를 순차적으로 곱하고, 그 결과를 저장하며, 합계를 계산한다.

파일명: scalar_main.c

```
#include <stdio.h>

// NASM에서 구현된 함수 선언
extern void scalar_vector_mul_sum(float *a, float *b, float *result,
float *sum);

int main() {
    float a[4] = {1.0, 2.0, 3.0, 4.0};
    float b[4] = {5.0, 6.0, 7.0, 8.0};
    float result[4];
    float sum = 0.0;

    // NASM 함수 호출
    scalar_vector_mul_sum(a, b, result, &sum);

    // 결과 출력
    printf("Result vector: [%f, %f, %f, %f]\n", result[0], result[1],
result[2], result[3]);
    printf("Sum: %f\n", sum);

    return 0;
}
```

NASM 어셈블리 코드: for문 기반 Scalar 연산

- NASM 어셈블리로 구현된 `scalar_vector_mul_sum` 함수는 반복문(`for`)을 활용해 어셈블리로 변환하여 스칼라 연산을 수행
  - 아래 비워있는 `TODO` 에 해당하는 명령어를 채우시오.

파일명: scalar_vector_mul_sum.asm

```
section .data
    zero dd 0.0                ; float 0.0


section .text
    global scalar_vector_mul_sum

; void scalar_vector_mul_sum(float *a, float *b, float *result, float
*sum)
scalar_vector_mul_sum:
    ; 레지스터 매핑
```

```
    ; a -> rdi, b -> rsi, result -> rdx, sum -> rcx

    ; 초기화
    xor r8, r8                  ; r8 = 0 (루프 카운터 i)
    movss xmm3, dword [rel zero] ; xmm3 = 0.0 (합계 초기화)

.loop_start:
    ; i가 4보다 크거나 같은지 확인
                                ; TODO: if (i >= 4) break;
    jge .loop_end

    ; a[i]를 xmm0에 로드
    mov rax, r8                 ; rax = i
                                ; TODO: rax = i * 4 (float 크기 계산), i
<<2로
    movss xmm0,                 ; TODO: xmm0 = a[i]

    ; b[i]를 xmm1에 로드
    movss xmm1, dword [rsi + rax] ; xmm1 = b[i]

    ; xmm0 * xmm1 (곱셈 수행)
    mulss xmm0, xmm1            ; xmm0 = xmm0 * xmm1

    ; result[i]에 저장
                                ; TODO: result[i] = xmm0

    ; 합계에 추가
    addss xmm3, xmm0            ; xmm3 += xmm0

    ; i 증가
    inc r8                      ; i++

    ; 루프 반복
    jmp .loop_start

.loop_end:
    ; 합계를 sum에 저장
                                ; TODO: *sum = xmm3

    ; 함수 종료
    ret
```

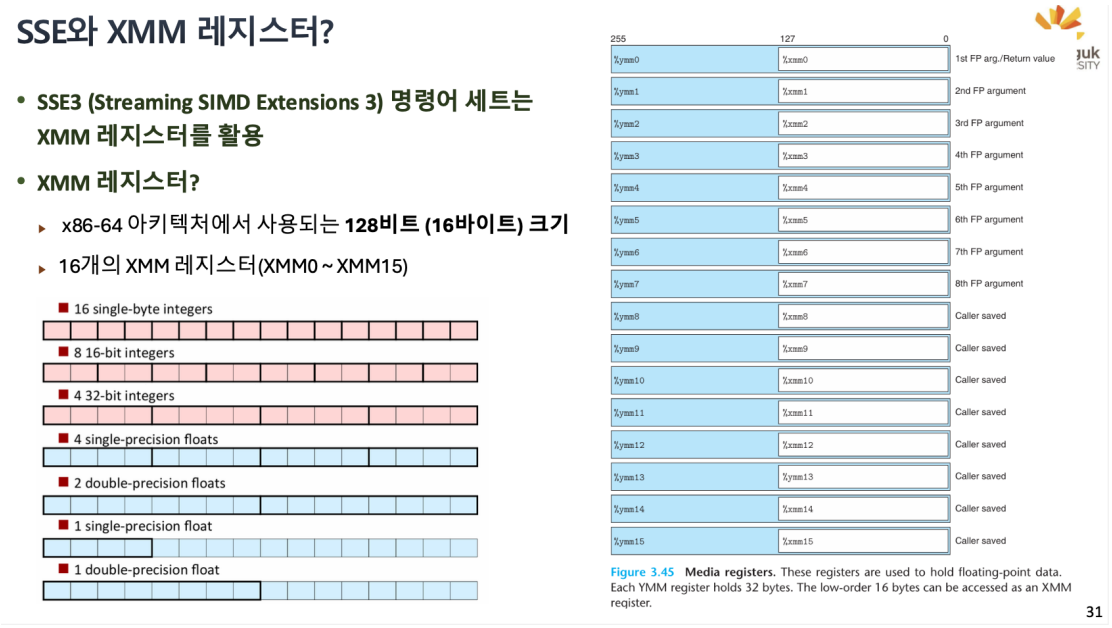
설명

- **어셈블리 코드 예시**

```
section .data
    zero dd 0.0  ; 단일 정밀도 부동소수점 값 0.0

section .text
    movss xmm3, dword [rel zero]  ; xmm3 = 0.0 (하위 32비트)
```

- **동작 설명**
  1. **데이터 정의**:
     - `.data` 섹션에서 `zero` 레이블은 4바이트 크기의 부동소수점 값 `0.0`을 저장
       - 여기서, `dd` : 32비트(4바이트) 크기의 데이터를 정의하는 지시자
         - 8바이트 지시자는 `dq`
  2. **데이터 로드**:
     - `.text` 섹션에서 `movss` 명령어는 `zero`의 값을 읽어와 `xmm3` **레지스터**의 하위 32비트에 로드
       - 결과적으로 `xmm3` **= [0.0 | 상위 96비트 기존 값], 총 128비트임을 상기하자**



컴파일 및 실행

1. NASM 코드 컴파일

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

2. C 코드 컴파일 및 NASM 오브젝트 파일 링크

```
gcc -m64 -o scalar_program scalar_main.c scalar_vector_mul_sum.o
```

3. 실행

```
./scalar_program
```

제출물:

- 어셈블리 구현물
- 실행 결과 (화면 캡쳐: 학번표시)

문제 2. Vector Operation

Test c언어: simd_main.c

```
#include <stdio.h>
// NASM에서 구현된 함수 선언
extern void simd_vector_mul_sum(float *a, float *b, float *result, fl
oat *sum);

int main() {
    // 데이터 초기화
    float a[4] = {1.0, 2.0, 3.0, 4.0};    // 첫 번째 벡터
    float b[4] = {5.0, 6.0, 7.0, 8.0};    // 두 번째 벡터
    float result[4];                      // 원소별 곱셈 결과 저장 배열
    float sum = 0.0;                      // 합계를 저장할 변수

    // NASM에서 작성된 SIMD 함수 호출
    simd_vector_mul_sum(a, b, result, &sum);

    // 결과 출력
    printf("Result vector: [%f, %f, %f, %f]\n", result[0], result[1],
result[2], result[3]);
    printf("Sum: %f\n", sum);

    return 0;
}
```

NASM 어셈블리 코드: simd_vector_mul_sum.asm

- 위와 마찬가지로 TODO에 해당하는 명령어를 채우시오

- 아래 설명 참조

```
section .text
    global simd_vector_mul_sum

; void simd_vector_mul_sum(float *a, float *b, float *result, float *
sum)
simd_vector_mul_sum:
    ; 레지스터 매핑
    ; a -> rdi, b -> rsi, result -> rdx, sum -> rcx

    ; 1. 벡터 데이터 로드
    vmovaps xmm0, [rdi]       ; rdi(a)에서 4개의 float 값을 xmm0으로 로드
    vmovaps xmm1, [rsi]       ; rsi(b)에서 4개의 float 값을 xmm1으로 로드

    ; 2. 원소별 곱셈
    vmulps xmm2, xmm0, xmm1   ; xmm2 = xmm0 * xmm1

    ; 3. 곱셈 결과를 result 배열에 저장
                              ; TODO: xmm2를 rdx(result) 메모리에 저장

    ; 4. 벡터 합계 계산 (수평 덧셈)
    vhaddps xmm2, xmm2, xmm2  ; xmm2의 4개 값을 수평 덧셈으로 합산
                              ; TODO: 두 번째 수평 덧셈으로 최종 합계 계산

    ; 5. 합계를 sum 변수에 저장
                              ; TODO: rcx(sum) 위치에 첫 번째 원소 저장

    ; 6. 함수 종료
    ret
```

코드 설명

- `vmovaps xmm0, [rdi]`: 배열 `a`의 4개 단일 정밀도(float) 값을 `xmm0` 레지스터에 로드.
- `vmovaps xmm1, [rsi]`: 배열 `b`의 4개 단일 정밀도(float) 값을 `xmm1` 레지스터에 로드.

`vmulps` 명령어 설명

`vmulps`는 **SIMD (Single Instruction Multiple Data)** 명령어 중 하나로, **패킹된 단일 정밀도 부동소수점 값**(packed single-precision floating-point values), 즉 `벡터 단일 정밀도 부동소수점`,을 처리하는 곱셈 연산 명령어

- 형식 : vmulps dest, src1, src2
  - `dest` **(Destination)**: 연산 결과를 저장할 목적지 레지스터

- `src1` **(Source 1)**: 첫 번째 입력 벡터 (레지스터나 메모리에서 가져옴)
- `src2` **(Source 2)**: 두 번째 입력 벡터 (레지스터나 메모리에서 가져옴)

- **작동 방식**
  - `src1`과 `src2`의 **각각의 대응되는 부동소수점 값들**을 곱하고, 결과를 `dest`에 저장
  - 연산은 병렬적으로 수행되며, **128비트(xmm 레지스터)** 또는 **256비트(ymm 레지스터)** 내의 여러 원소를 동시에 처리

예: 128비트 레지스터 (xmm)

src1 (xmm0)	src2 (xmm1)	dest (xmm2)
a0	b0	a0 * b0
a1	b1	a1 * b1
a2	b2	a2 * b2
a3	b3	a3 * b3

- `vmovaps [rdx], xmm2`: 곱셈 결과를 `result` 배열 메모리에 저장.

4. 수평 덧셈

- `vhaddps xmm2, xmm2, xmm2`: `xmm2`의 원소를 두 개씩 더한다.
- 두 번 반복하여 벡터 전체의 합계를 계산.

`vhaddps` 명령어 설명

당 코드는 SIMD 명령어를 활용해 벡터 원소의 **합계**를 계산하는 부분

- 주요 작업은 `vhaddps` 명령어를 사용한 **수평 덧셈(horizontal addition)**을 통해 벡터 원소를 반복적으로 더하고, 최종 합계를 메모리에 저장하고자 함.

---

1. `vhaddps xmm2, xmm2, xmm2`

- `vhaddps`는 **Packed Single-Precision Floating-Point Horizontal Add** 명령어로, 벡터 레지스터의 **짝수-홀수** 쌍을 수평으로 더
- 연산은 SIMD 레지스터 내에서 병렬적으로 수행된다.
- 결과는 동일한 레지스터(`xmm2`)에 저장된다.

작동 방식

초기 상태 (`xmm2`의 값)

벡터가 아래와 같은 값을 가지고 있다고 가정:

```
xmm2 = [a0, a1, a2, a3]  ; 4개의 32비트 float 값
```

첫 번째 수평 덧셈: `vhaddps xmm2, xmm2, xmm2`

- 레지스터 내부의 값을 **짝수-홀수** 쌍으로 더함.
- 결과:

```
xmm2 = [a0 + a1, a2 + a3, ?, ?]
```

- 예를 들어:

```
xmm2 = [1.0, 2.0, 3.0, 4.0]
       → [1.0 + 2.0, 3.0 + 4.0, ?, ?]
       → [3.0, 7.0, ?, ?]
```

- 여기서 `?`는 더 이상 사용되지 않는 값이며 무시된다.

두 번째 수평 덧셈: `vhaddps xmm2, xmm2, xmm2`

- 다시 수평으로 더한다:

```
xmm2 = [3.0 + 7.0, ?, ?, ?]
       → [10.0, ?, ?, ?]
```

- 이제 레지스터의 첫 번째 원소에 전체 벡터의 합계가 저장

컴파일 및 실행

1. NASM 코드 컴파일

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

2. C 코드 컴파일 및 NASM 오브젝트 파일 링크

```
gcc -m64 -o simd_program simd_main.c simd_vector_mul_sum.o
```

3. 실행

```
./simd_program
```

제출물:

- 어셈블리 구현물
- 실행 결과 (화면 캡쳐: 학번표시)

- **SIMD와 스칼라 방식 연산 비교 (짧게 2줄로 작성)**

문제 3. 비교 명령어 실습: `ucomiss` 와 `ucomisd` 활용

- 다음은 부동소수점 비교 명령어 `ucomiss`(float)와 `ucomisd`(double)를 사용하여 두 숫자를 비교하고 조건 코드를 사용해서 return 값을 조정해보자

목표

1. `ucomiss`와 `ucomisd` 명령어를 사용해 두 부동소수점 숫자를 비교.
2. 조건 코드(CF, ZF, PF)를 설정하고 이를 사용해 특정 작업 수행.
3. 비교 결과에 따라 메시지를 출력.

C 코드: comp_main.c

```
#include <stdio.h>

// NASM에서 구현된 함수 선언
extern int compare_floats(float a, float b);
extern int compare_doubles(double a, double b);

int main() {
    float f1 = 5.5, f2 = 3.3;
    double d1 = 4.4, d2 = 4.4;

    // float 비교
    int float_result = compare_floats(f1, f2);
    printf("Float comparison result: %d\n", float_result); // -2, 1,
0, 1 중 하나 출력

    // double 비교
    int double_result = compare_doubles(d1, d2);
    printf("Double comparison result: %d\n", double_result); // -2, -
1, 0, 1 중 하나 출력

    return 0;
}
```

- **`compare_floats` 함수**
  - 두 `float` 값을 비교하고, 다음 결과를 반환:
    - `1`: 첫 번째 값이 작음.

    - `0`: 두 값이 같음.
    - `1`: 첫 번째 값이 큼.
    - `-2`: NAN 값을 연산하면?
- `compare_doubles` **함수**
    - 두 `double` 값을 비교하고 동일한 결과를 반환.

NASM 어셈블리 코드: compare.asm

- 마찬가지로 **TODO** 명령어를 채우시오.

```
nasm
Copy code
section .text
    global compare_floats
    global compare_doubles

; int compare_floats(float a, float b)
compare_floats:
    ; 매개변수: a -> xmm0, b -> xmm1
                              ; TODO: float a와 float b 비교
    jp .nan_case              ; NaN일 경우 처리
    jb .less                  ; a < b: CF = 1
    ja .greater               ; a > b: CF = 0, ZF = 0
    je .equal                 ; a == b: ZF = 1

.less:
    mov eax, -1               ; 반환 값: -1 (a < b)
    ret

.greater:
                               ; TODO: 반환 값: 1 (a > b)
    ret

.equal:
                               ; TODO: 반환 값: 0 (a == b)
    ret

.nan_case:
                               ; NaN 경우: 반환 값 -2
    ret

; int compare_doubles(double a, double b)
compare_doubles:
```

```
    ; 매개변수: a -> xmm0, b -> xmm1
                              ; TODO: double a와 double b 비교
    jp .nan_case_d            ; NaN일 경우 처리
    jb .less_d                ; a < b: CF = 1
    ja .greater_d             ; a > b: CF = 0, ZF = 0
    je .equal_d               ; a == b: ZF = 1

.less_d:
                              ; TODO: 반환 값: -1 (a < b)
    ret

.greater_d:
                              ; TODO: 반환 값: 1 (a > b)
    ret

.equal_d:
                              ; TODO: 반환 값: 0 (a == b)
    ret

.nan_case_d:
                              ; TODO: NaN 경우: 반환 값 -2
    ret
```

코드 설명

1. `ucomiss` 와 `ucomisd`
   - `ucomiss xmm0, xmm1`: 단일 정밀도(float) 비교.
   - `ucomisd xmm0, xmm1`: 배정밀도(double) 비교.
   - 두 명령어 모두 비교 결과를 조건 코드(CF, ZF, PF)에 설정.
2. **조건 코드 사용**
   - `CF = 1`: 첫 번째 값이 두 번째 값보다 작음.
   - `ZF = 1`: 두 값이 같음.
   - `PF = 1`: NaN(Not a Number)인 경우.

`jp` 명령어에 대한 설명

- `jp`는 x86 어셈블리 명령어로, **"Jump if Parity"**를 의미
- 이 명령어는 조건부 점프 명령어 중 하나로, **Parity Flag (PF)**의 상태에 따라 점프를 수행

`jb`: Jump if Below

- **의미**: "첫 번째 값이 두 번째 값보다 작은 경우 점프."

- **조건**: **Carry Flag (CF)**가 **1**일 때 점프

`ja` : Jump if Above

- **의미**: "첫 번째 값이 두 번째 값보다 큰 경우 점프."
- **조건**: Carry Flag (CF)가 **0**이고 Zero Flag (ZF)가 **0**일 때 점프
- 플래그 상태:
  - `CF = 0`: 첫 번째 값이 두 번째 값보다 작지 않음.
  - `ZF = 0`: 두 값이 같지 않음.
  - 따라서, 두 값 중 첫 번째 값이 두 번째 값보다 클 경우.

`je` : Jump if Equal

- **의미**: "두 값이 같은 경우 점프."
- **조건**: **Zero Flag (ZF)**가 **1**일 때 점프.

입력값:

- `f1 = 5.5, f2 = 3.3`
- `d1 = 4.4, d2 = 4.4`

컴파일 및 실행

1. NASM 어셈블리 파일 컴파일

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

2. C 코드 컴파일 및 NASM 오브젝트 파일 링크

```
gcc -m64 -o compare_program compare_main.c compare.o
```

3. 실행

```
./compare_program
```

제출물:

- 어셈블리 구현물
- 실행 결과 (화면 캡쳐: 학번표시)

💡 최종 보고서에는 문제 1,2,3에 대한 구현물, 실행 결과, 문제 2번 같은 경우에 대해서는 비교에 대한 생각을 짧게 2줄로 정리해서, 하나의 pdf로 제출해주세요