CSED211 Homework3

20210774 김주은

**1. Practice Problem 3.46 on Page 318.**

In C code, ‘buf[4]’ is changed to ‘buf[8]’. In disassembly, ‘sub $0x10, %rsp’ is changed to ‘sub ‘$0x18, %rsp’. Input string is “0123456789012345678901234567890123”

A.

Getline의 첫번째 줄부터 실행이 되면 push %rbx가 진행되면서 %rsp값은 8만큼 감소하고, stack의 top에 %rbx에 들어 있는 값인 0123456789ABCDEF가 들어간다.

이후 2번째 줄에서 %rsp값을 총 0x18, 즉 24만큼 감소시키므로, 이를 통해 3번째 줄이 실행되고 난 직후의 rsp가 가리키는 곳과 stack의 구조를 알 수 있게 된다.

테이블이(가) 표시된 사진

자동 생성된 설명

B.

Input string is “0123456789012345678901234567890123” 이고, 문자열이므로 ascii코드 값으로 입력되며 마지막에 33이 들어간 후 null character도 끝에 들어간다. 이를 %rsp부터 차례대로 넣으면 아래와 같은 stack구조 형태를 띈다.

테이블이(가) 표시된 사진

자동 생성된 설명

C.

Return address가 0x003332로 바뀌었으므로, 0X003332 주소로 리턴할 것이다.

D.

Stack구조에서 보면 기존에 %rbx의 값이 저장되어 있던 곳에 입력값들이 쓰여지면서 값들이 31 30 39 38 37 36 35 34로 수정되었다. %rbx는 callee-saved register이므로 함수가 시작할 때 push %rbx, 끝날 때 pop %rbx를 한다. 하지만 pop %rbx를 할 때 기존의 rbx값이 아닌 31 30 39 38 37 36 35 34이 %rbx 레지스터에 저장되므로 결론적으로 get\_line이 return하면서 %rbx값이 변하게 된다.

E.

- Malloc을 호출하는 부분에서 문자열에 대해 공간을 할당하기 때문에 Null character을 고려하여 strlen(buf)+1으로 할당을 해야한다. 즉, result = malloc(strlen(buf)+1)이 되어야 한다.

- malloc 호출 후 return할 시 리턴값이 result에 저장되는데 이때 malloc의 리턴값이 null인지 아닌지도 확인하는 코드가 필요하다.

**2. Practice Problem 3.47 on Page 322**

A. Approximate range with the addresses ranged from 0x7FFF7F01B680 to 0x7FFFFFA53B40

0x7FFFFFA53B40 - 0x7FFF7F01B680 = 0x80A384C0

0x80A384C0은 십진수로 2158200000이므로, 총 2158200000만큼의 range를 가진다.

B. How many attempts with 256-byte nop sled?

2158200000에서 2^8을 나누면 8430468.75이므로, 총 8430469번의 시도를 하면 된다.

**3. Exercise 5.13 on page 606**

A.

텍스트이(가) 표시된 사진

자동 생성된 설명

이 루프부분의 어셈블리 코드

텍스트이(가) 표시된 사진

자동 생성된 설명

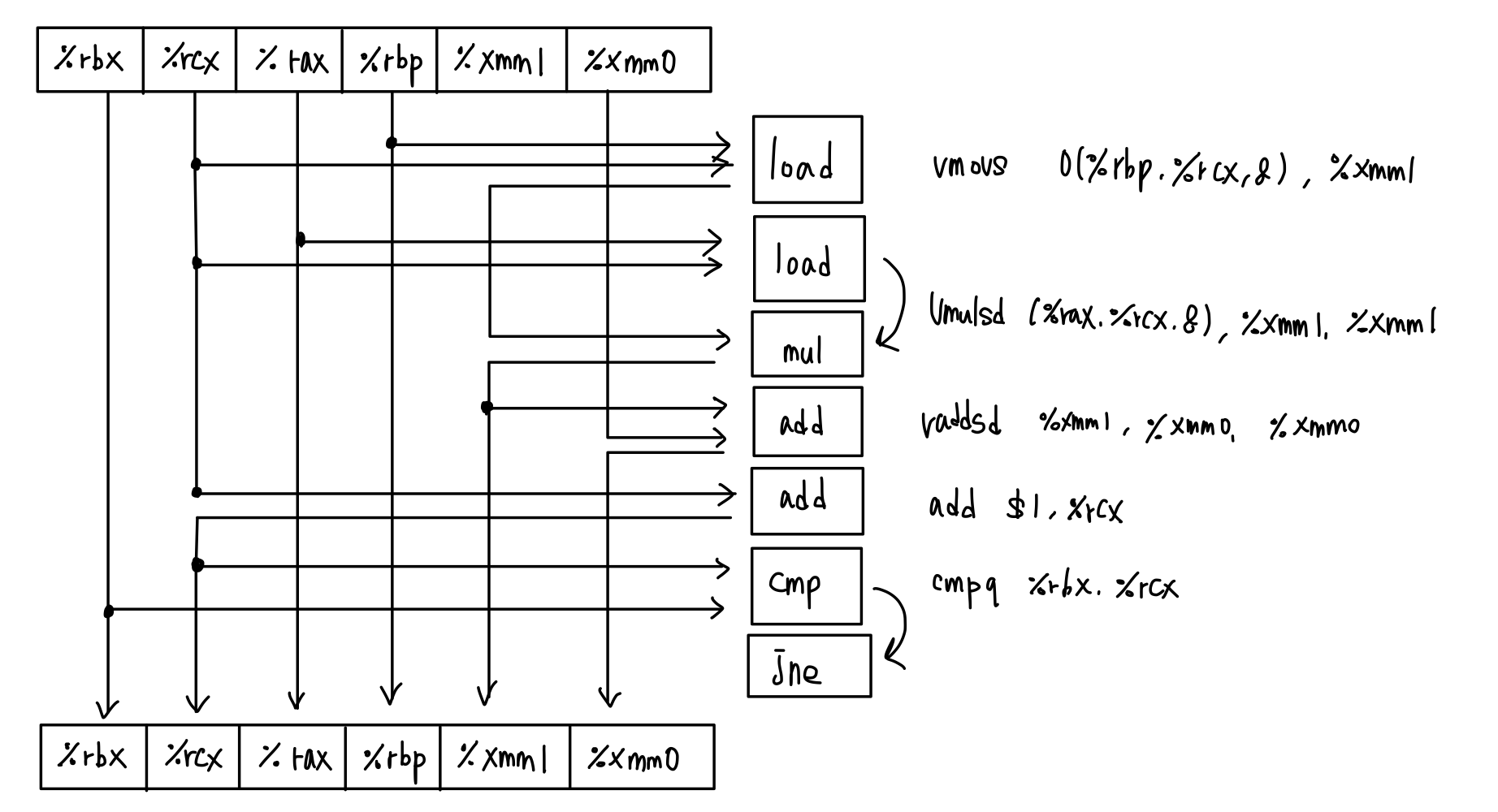
Rbx 레지스터의 값의 경우 limit값이며 이는 loop를 돌면서 변하지 않는 값이고, cmp인스트럭션의 소스로 한번 쓰인다.

Rcx 레지스터의 값은 udata[i]와 vdata[i] 의 값들이 load될때 같이 load되며, rcx를 1만큼 add해주는 인스트럭션을 통해 업데이트되어 값이 한 번 갱신된다. 그리고 cmp instruction의 소스로도 쓰인다.

Rax와 rbp는 각각 udata와 vdata값을 가지고 있으며 루프를 도는 동안 값이 갱신되지 않고 변하지않는다. Rcx와 마찬가지로 udata[i]와 vdata[i] 값들이 load될 때 load instruction의 소스로 쓰인다.

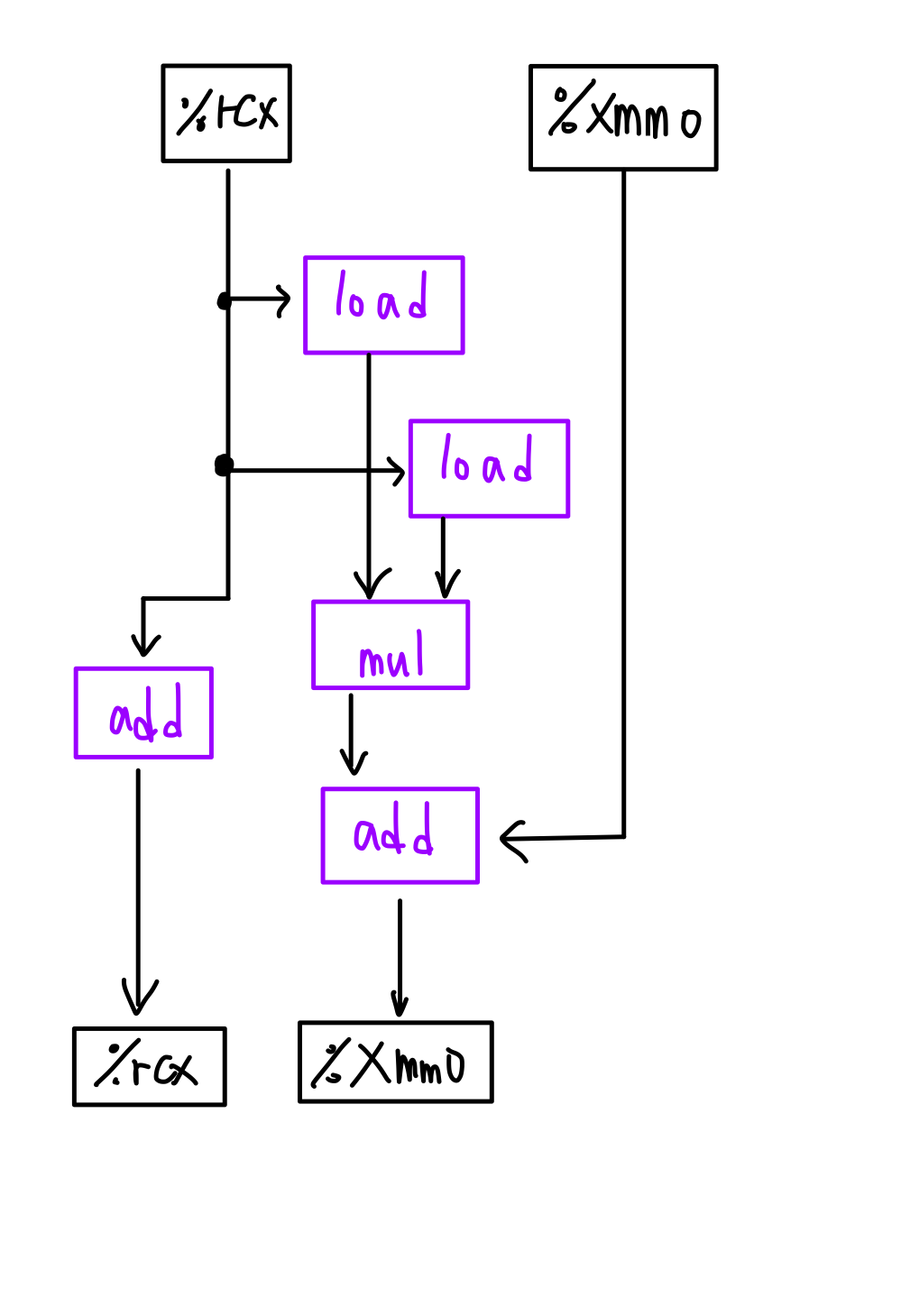
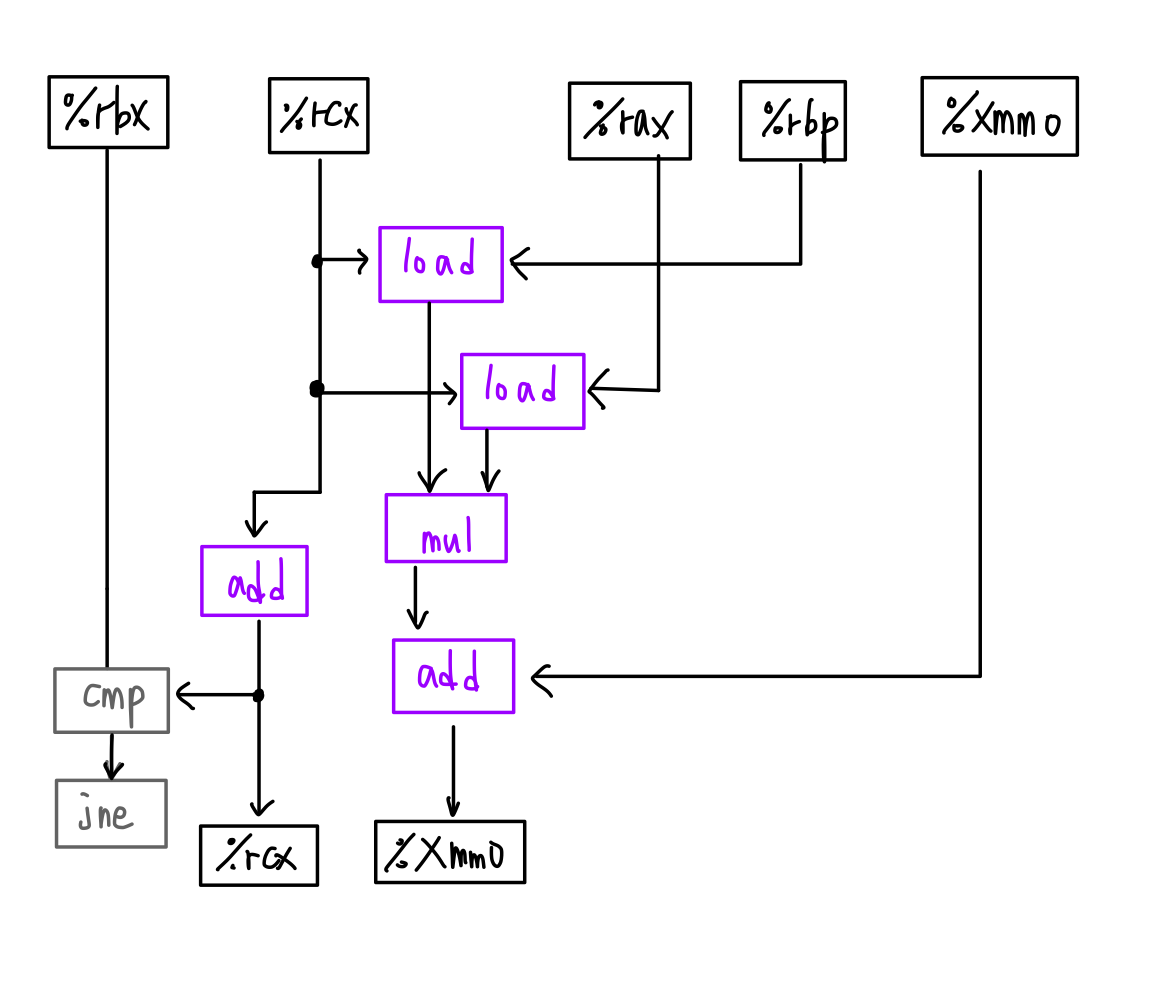
Xmm1의 경우 vdata를 load할때 load instruction의 결과값이 저장된 후, udata를 load하여 곱할때 소스와 목적지 둘다로 쓰인다. 이렇게 load와 mul을 통해 값이 갱신된 이후 xmm0와 더하는 add instruction에서 소스로 한번 더 쓰인다.

Xmm0의 경우 xmm1와 더하는 add instruction에서 소스와 목적지로 쓰이고, 이 과정을 통해 값이 갱신된다. 이를 바탕으로 그림을 그리면 다음과 같다.



위의 그림을 바탕으로,

데이터 의존성을 잘 보여주는 그림을 그렸다. 그리고 오른쪽 그림은 다음 반복 실행을 위한 새로운 값들을 생성하기 위해, 한 번 반복 실행에서 나온 값들을 사용하는 연산들 만을 뽑아서 다시 그린 것이다.



B.

위의 그림에서 데이터의존성을 살펴보면, load 연산과 mul연산의 경우에는 %xmm0의 add연산과 독립적으로 계산이 가능하기에 곱해진 xmm1값과 xmm0 사이의 부동소수점 addition을 하는 부분을 중심으로 key path를 이룬다.

그러므로, CPE의 최저 경계값은 floating point addition의 latency인 3.00이다.

C.

만약, 부동소수점이 아니라 정수 데이터형을 사용하더라도 key path의 구조는 동일할 것이다. 정수의 경우에는 정수 addition의 latency인 1.00이 될 것이다.

D.

Floating-point의 multiplication operation은 5 clock cycle을 요구하지만, multiplication operation은 key path에 포함되어 있지 않다. 즉, 이전 값의 계산이 끝난 후 직렬적으로 multiplication이 진행되지 않고, 독립적으로 진행될 수 있기에 총 total clock cycle이 5\*n의 형태로 나오지 않는다. 위에서 설명했던 것처럼 key path를 이루는 floating-point addition operation만 직렬적으로 진행되기에 CPE는 3.00이 되는 것이다.

4. Exercise 5.19 on page 609

In problem 5.12,

텍스트이(가) 표시된 사진

자동 생성된 설명

이런 형태를 띈 Psum1이라는 함수가 주어졌을 때, memory에서 p[i]값을 계속하여 가져오지 않아도 되도록 만드는 것이 문제 5.12의 내용이었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 해당 문제의 결과는 위와 같다. 그리고 이때 3.00으로 CPE를 줄일 수 있었다.

CPE를 더 줄이기 위하여 loop unrolling 과 reassociation을 통해 줄여 나가야 할 것이다.

Loop unrolling과 reassociation을 통해 코드를 재구성하면 Loop안에 하나의 element가 아닌 여러 element에 대해 덧셈 연산이 이루어져 있을 것이다.

문제에서 주어진 조건대로 CPE = 1.67 이고, 이는 분수꼴로 나타내면 5/3임을 알 수 있다.

즉, unlolling by 3을 하고, loop안에 덧셈 연산 5개가 있으면 될 것이다.

\*\*이를 반영한 c코드를 c파일에 첨부하였다.

**5. Exercise 6.24 on page 685**

텍스트이(가) 표시된 사진

자동 생성된 설명

파일이 2MB이고, logical block 하나당 512byte이므로, 총 4000개의 logical block으로 이루어진 파일이라고 할 수 있다.

T\_avg\_seek = 8이고,

T\_avg\_rotation =

T\_max\_rotation = 2\*T\_avg\_rotation = 3.34

T\_avg\_transfer = T\_max\_rotation / 2000 = 0.00167

A.

최적의 경우에는 헤드를 움직이지 않고 하나씩 연속하여 읽을 수 있는 섹터들로 매핑이 되는 경우이다. 즉, 4000개의 logical block에 대응되는 섹터들이 연속적으로 존재하면 된다. 한 트랙당 평균적으로 2000개의 sector가 존재하므로 총 2회 회전하면 파일을 다 읽을 수 있을 것이다.

그러므로, 총 파일을 읽는 데 걸리는 시간은 (헤드를 첫번째 블록 위로 위치하는 데 걸리는 시간) + 2번 회전하는 시간이다.

헤드를 첫번째 블록 위로 위치하는 데 걸리는 시간은 문제에서 주어진 대로 T\_avg\_seek + T\_avg\_rotation 이므로 8 + 1.67 = 9.67 이다.

2번 회전하는 시간의 경우 = 2\*T\_max\_rotation = 2\*3.34 = 6.68이다.

Answer : 16.35ms

B.

랜덤의 경우에는 블록들이 랜덤으로 섹터에 매핑되기 때문에 logical block 하나를 읽기 위해서는 T\_avg\_seek + T\_avg\_rotation + T\_avg\_transfer 만큼이 걸린다.

T\_avg\_seek은 8ms이고, T\_avg\_rotation은 1.67이고, T\_avg\_transfer은 0.00167이다.

총 세개를 더한 값은 9.67167이다.

총 block수는 4000개이므로, 파일을 읽는데 총 걸리는 시간은 9.67167\*4000 = 38686.68이다.

Answer : 38.69ms