어셈블리 프로그래밍 설계 및 실습

실험제목: Multiplication of Floating Point

<Project 최종보고서>

제출일자: 2017년 11월 29일 (수)

학 과: 컴퓨터공학과

담당교수: 이형근 교수님

실습분반: 월 5, 수 6, 7

학 번: 2014722075

성 명: 이 동 준

1. Introduction

O 일정

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 11월 | | | | | | |
| 12 | **13** | **14** | **15** | **16** | **17** | **18** |
| 제안서 작성 및 프로젝트 해석 | | | | 세부설계 - Addition(+Subtraction) of Floating Point Number | | |
| 19 | **20** | **21** | **22** | **23** | **24** | **25** |
| 세부설계 – Multiplication of Floating Point Number(Booth Multiplication with Radix – 4) | | | 세부설계 – 정방행렬 곱셈 구현 및 addition and multiplication 검증, 최종보고서 작성. | | | |
| 26 |  |  |  |  |  |  |
| 최종보고서 제출 |  |  |  |  |  |  |

2. Project Specification

- Floating Point Number의 Multiplication을 구현한다.

우선 Floating Point Number는 32bit에 Sign Bit(1bit), Exponent(8bits), Mantissa(23bits)로 표현된 숫자이며, sign bit는 값의 부호, exponent는 2의 제곱수 + 127, mantissa는 그 값의 소수점을 나타낸다.



# sign bit : 양수인지 음수인지 구분을 해준다.(0 – 양수, 1 – 음수)

# Exponent : 2진수의 수를 나타내기 위해 지수를 이용.

# Mantissa : Fraction에서 소수점 왼쪽을 제외한 수.

-> (-1)S \* (1.mantissa) \* 2(exponent – 127)

더불어 NaN(Not A Num)과 (+-)Infinity, 0 을 구분하여 연산할 수 있다.

- Multiplication method – Radix – 4

floating point num의 곱셈연산을 위해 이용한 방법은 Booth Multiplication이다.



-> sign bit – 두 수의 sign bit를 XOR연산을 통하여 결과 값의 sign bit를 결정 -> S1⊕S2

-> exponent – 두 수의 exponent는 실질적인 값 +127로 나타나기 때문에 두 exponent를 덧셈연산 후 -127을 통해 결과 값의 exponent를 결정 -> 2(exponent1 + exponent2 – 254)

-> mantissa – 두 수의 mantissa를 fraction으로 변경 후 multiplier와 multiplicand을 설정하여 Radix – 4 방식으로 곱셈연산을 진행한다.

-> 정방행렬의 곱셈(30x30) – 각 900개의 데이터를 가지고 있는 행렬 두 개를 곱셈연산을 실시 한다. 결과 행렬은 900개의 데이터를 가지고 있다.

3. Algorithm

main

r3 = 0, count, count2 = 0

while(count2 != 27000)

r5 = load multiplicand

r0 = move pointer1 to next address of value

r6 = load multiplier

r2 = move pointer2 to next address of value

r7 = multiplication(r5,r6)

//r5 is multiplicand, r6 is multiplier, r7 is result of multiplication

r3 = addition(r3,r7) //r3 will be 1 result of result matrix

count++ //counter 900 -> 결과행렬의 한 행의 연산횟수

count2++ //counter 27000 -> 결과행렬의 전체 연산횟수

if(count == 900)

if(infinity num == -infinity num)

store 0 in result matrix

else

store r3 in result matrix

r3 = 0

count = 0

//count2 == 27000

endprogram

multiplication(r5,r6)

if(r5 == NaN || r6 == NaN)

return NaN // return to main

if(r5 == 0 || r6 == 0)

return 0 // return to main

if(r5 == infinity || r6 == infinity)

if(r5 == -infinity || r6 == -infinity) //not same sign bit

return -infinity // return to main

return infinity // return to main

if(r5 == -infinity || r6 == -infinity)

if(r5 == infinity || r6 == infinity)

return infinity // return to main

return -infinity // return to main

make 2 sign bit

if(same sign bit)

r7 = 0(sign bit)

else

r7 = 1(sign bit)

make 2 exponent

r7 = r7 + (add 2 exponent-127)

make 2 fraction

assign(radix – 4) r8 = fraction1 \* fraction2

r7 = r7 + r8

return r7 // return to main

addition(r3,r7)

if(r3 == 0)

return r7

if(r7 == 0)

return r3

if(r3 == NaN || r7 == NaN)

return NaN

if(r3 == infinity || r7 == infinity)

if(r3 == -infinity || r7 == -infinity)

return 0

return infinity

if(r3 == -infinity || r7 == -infinity)

if(r3 == infinity || r7 == infinity)

return 0

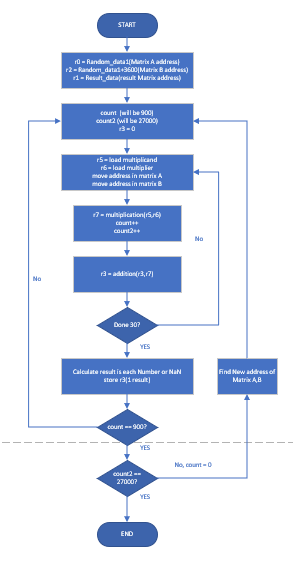
return -infinity

compare absolute value

assign large absolute value – small

sign bit is sign bit of large absolute value

return result



4. Performance & Result

\* Performance

code size : 10580 Bytes , state : 9398842 (case 1)

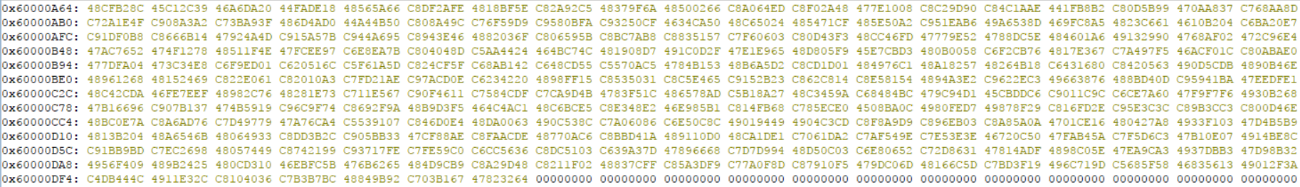
: 9167625 (case 2)

: 8760712 (case 3)

\* Result

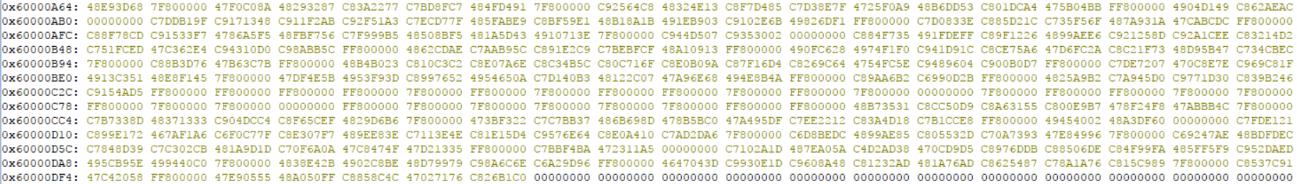
(case 1)



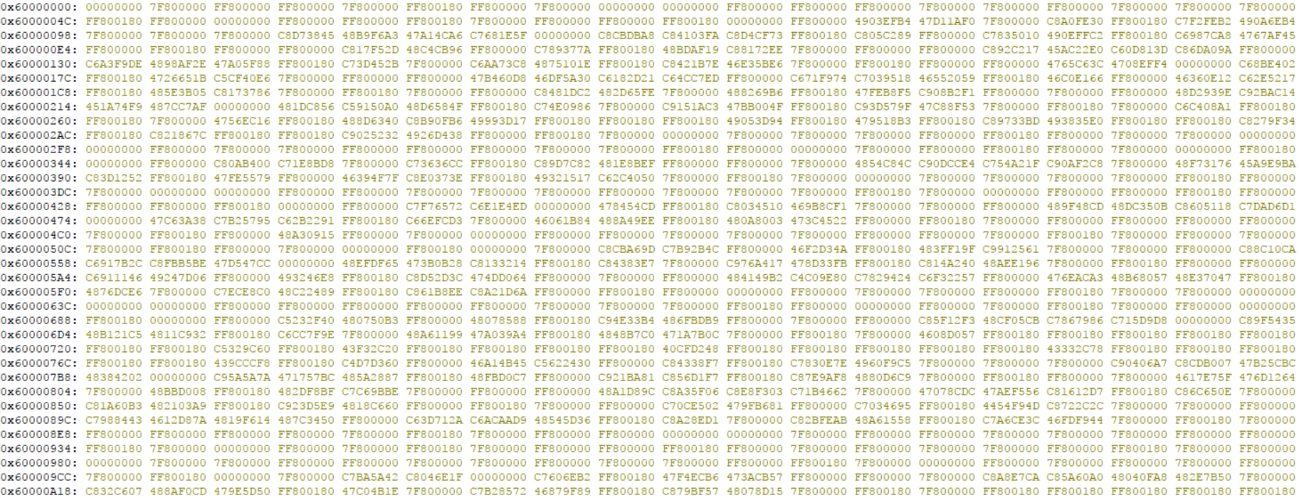


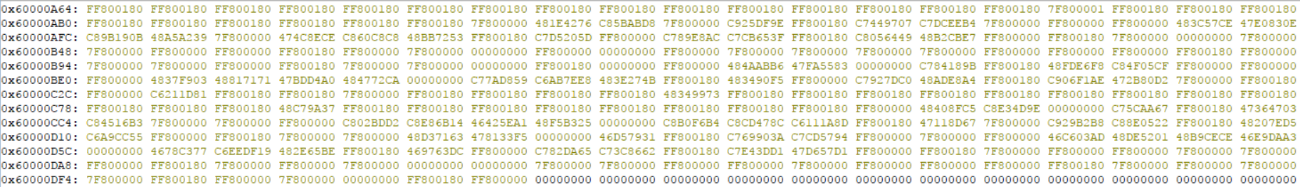
(case 2)





(case 3)





5. Conclusion

이번 프로젝트에서는 Floating Point Number의 곱셈에 대하여 설계하였다. 이전 실습에서 Floating Point Number의 덧셈에 대해서 다루었고, 그 실습 내용을 기반하여 곱셈을 진행하였다. 32bit에서 나타나는 각 bit들의 의미에 대해 정확히 분석하고, MUL 명령어가 아닌 Booth Multiplication 방식을 통하여 곱셈을 진행하였다. floating point에서 실질적인 값을 가지는 부분인 mantissa(fraction)에 대한 곱셈을 booth multiplication 방식으로 진행하였고, sign bit나 exponent는 곱셈에 맞게 설정하여 설계를 하였다. 제안서를 읽으면서 컴퓨터기초설계 및 실험에서 사용하였던 booth multiplication을 이용하기에 쉽게 완성을 할 것이라 예상을 하였지만, fraction(24bit)과 fraction의 곱셈의 결과는 48bit이기 때문에 mantissa의 23bit 안에 표현하기란 쉽지 않았다. 또한 정확한 연산 결과가 아닌 근사치가 나왔기 때문에 여러가지 방법을 통해서 해결하려 했다. 더불어 special case에 대한 관리도 고충을 겪었다. 생각보다 값을 마음대로 만들고 비교하기 힘들었고, state를 최소로 하자 라는 생각을 하다 보니 더욱 막막하였다. 또한 30x30 정방행렬의 곱셈을 진행해야 하기 때문에 메모리 주소 관리에 있어서도 고민을 많이 하였다. 다양한 방법을 시도한 결과 기대했던 결과가 나왔고, 이번 프로젝트를 통해 floating point number에 대해 좀 더 자세히 알게 되었고, booth multiplication의 radix – 2와 radix – 4를 모두 시행하면서 프로그램의 성능 차이에 대해서도 이해도가 상승하였다.