

어셈블리프로그램 설계 및 실습

Term Project 보고서



담당교수	이형근 교수님
실습분반	화요일 (2분반)
소속	컴퓨터정보공학부
학번·성명	2023202027 정정윤
제출일	2024년 12월 09일

1. Problem Statement

IEEE - 754 방식으로 표현된 자료를 FPU가 포함되지 않은 ARM 프로세서에서 처리하는 방법을 설계하고, 작성하여 실행하고, 그 효율성과 정밀도를 확인한다.

ARM 프로세서에 대해 Keil 시뮬레이터를 작동시키며, 작성한 코드의 작동 과정과 작동 시간 등을 확인하여, 각 명령과 프로그램의 효율성을 확인한다.

과제 목표는 20 * 20 개의 픽셀에 대해 각 픽셀의 활성화 정도를 나타낸 입력값을 Bilinear Interpolation을 통해 80 * 80 크기의 출력값을 내보내는 것이다.

입력값은 모두 IEEE - 754 Single Precision으로 표현된 값이 제공되며, 출력은 메모리에 진행한 후, 이를 SAVE 명령으로 hex파일로 내보내어 확인한다.

입력 데이터는 출력 데이터의 [0, 0]위치를 기준으로 4행/열 마다 배치되며, 가장 오른쪽과 가장 아래에 남는 3열, 3행을 padding으로 한다. Padding의 경우 가장 가까운 데이터를 복사하는 Nearest Padding을 사용한다.

정수로 바꾸어 연산해서는 안되며, IEEE - 754로 제공된 데이터 바탕으로 연산하여야 한다.

입력값은 0 또는 양수인 실수값이 들어오며, 음수값은 들어오지 않는다.

코드의 크기, 코드의 실행 시간은 직접적 평가 기준에는 들어가지 않지만, 추가 점수 요인으로 사용되므로, 코드 길이 및 시간의 최적화를 진행할 필요가 있다.

2. Algorithm

2-1. 서브루틴

IEEE 754 방식으로 표현한 부동소수점수를 FPU없이 연산하여야 하는 상황에 20*20 반복 및 각 보간 값의 저장을 동시에 수행하는 것은 레지스터 개수에 무리가 있으므로, 서브루틴을 만들어 함수처럼 사용하는 것을 기반으로 구현하였다.

두 부동소수점수의 합연산이나 곱연산을 수행할 때에는, 해당 서브루틴으로 이행 전, 매개변수로서 전달하기 위해 R0과 R1에 피연산 값들을 저장하고 전달한다. 이후 이외의 레지스터를 스택에 저장해두고, R0에 연산 결과를 반환하며, 다시 스택에서 꺼낸다.

반환값이 없는 경우에는 R0-R12를 모두 저장해도 무관하다. 20*20 영역에서 반복하는 구조와 이 중 4개의 값을 바탕으로 보간을 수행하는 루틴 간의 이동에서는 입력은 일어나지만

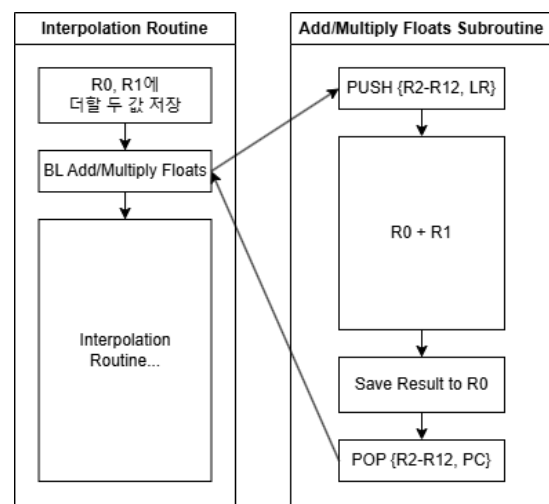


Figure 1. 서브루틴 구현 예시

반환은 일어나지 않는다. 작성한 코드에서 이는 반영되어있지 않지만, 보간 서브루틴으로의 이동을 수행하는 20*20 순회 루틴에서는 R0과 R1을 사용하고 있지 않기 때문에 스택의 공간을 조금 더 적게 사용할 뿐 차이는 발생하지 않는다.

2-2. 보간 루틴

23	63.25	103.5	143.75	184
72.5	104.8125	137.125	169.4375	201.75
122	146.375	170.75	195.125	219.5
171.5	187.9375	204.375	220.8125	237.25
221	229.5	238	246.5	255

Figure 2. 보간 예시

위 이미지는 20*20 to 80*80 보간 수행 과정에서 한 부분을 가져온 것이다. 보간 루틴에서는 가까운 4개의 값을 바탕으로 보간으로 수행하므로, 가장 가까운 4개의 값 이외의 값은 요구되지 않는다. 작성한 보간 루틴에서는 붉은 부분이 포함된 행과 열을 제외한 4*4 영역이 메모리에 채워지게 된다. 이 루틴은 입력으로써 R2-R5에 V1(주황색 칸), V2(초록색 칸), V3(파란색 칸), V4(보라색 칸)의 값을, R1에 저장 주소를 입력받는다.

입력 이후에는 입력 데이터를 바탕으로 4*4를 순회하며, 메모리에 값을 저장한다.

$$V_{ij} = \left(1 - \frac{i}{4}\right) \left(\left(1 - \frac{j}{4}\right) V_1 + \frac{j}{4} V_2 \right) + i \left(\left(1 - \frac{j}{4}\right) V_3 + \frac{j}{4} V_4 \right)$$

값을 토대로 메모리에 값을 저장하게 되며, 필요 레지스터 개수와 연산을 최소화하기 위해 $\frac{i}{4}$ 값과 $\frac{j}{4}$ 값의 경우, 미리 DCD로 0, 0.25, 0.5, 0.75, 1 값을 저장하여 사용하였다.

시행마다 $1 - \frac{n}{4}$ 를 계산하면 정수 → 부동소수점수 변환 연산 및 부동소수점수 곱연산, 부동소수점수 합연산을 수행해야 하기에 State가 크게 늘어난다.

2-3. 20*20 순회 루틴

20*20 순회와 보간 루틴을 분리하여, Padding의 연산에서도 보간 루틴을 그대로 사용할 수 있다. 우측 Padding의 경우 V1과 V3의 값을 일치시켜 Nearest Padding을 처리할 수 있으며, 하단 Padding의 경우 V1과 V3 값을, V2와 V4의 값을 일치시켜 Nearest Padding을 처리할 수 있다. 우측 하단의 경우 모든 V값을 V1값과 일치시켜 처리할 수 있다.

오른쪽 이미지는 20*20 순회 루틴의 순서도이다.

<https://drive.google.com/file/d/1uM5JYYImGQVZnqYAID-6Kx4ZlvghaMy/view?usp=sharing>

본 프로젝트 순서도 모음에서 상세 이미지를 확인할 수 있다.

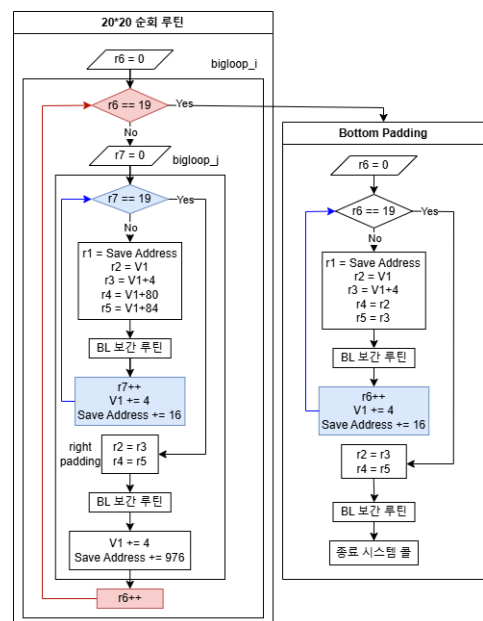


Figure 3. 20*20 루틴

2-4. 곱연산

IEEE 754 부동소수점수 간의 곱연산은 multiply_floats 서브루틴에서 수행한다. 곱연산은 align 과정의 불필요와 normalize의 간소화로 합연산보다 수행해야할 과정이 적다.

연산 시, Sign, Exponent, Fraction부분을 분리하고, 연산을 수행한다.

부호는 두 수의 Sign Bit간 EOR연산으로 수행한다. 곱연산이기에 부호가 같은 경우 Sign은 0, 다른 경우 Sign은 1로 한다.

Exponent끼리 더한 후, Exponent의 bias인 127을 뺀다. Exponent는 기본적으로 127이 더해진 수치이기에 두 Exponent를 더하면 bias가 두 번 적용된 것과 같으므로, 한 번 빼주어야 정상적인 값으로 출력된다. 곱연산이기에 지수인 Exponent부는 더하여야 한다.

Mantissa값의 곱연산을 수행한다. 24bit 수치 두 개의 연산이기에 항상 Overflow가 일어나므로, UMULL 명령을 통해 두 레지스터에 값을 저장한다. 이후 연산값 이후 Overflow된 값이 16비트보다 크다면 Exponent를 +1, 그렇지 않다면 대신 곱연산 결과를 LSL한다. 곱연산에서의 Normalize과정과 같다.

이후 연산된 값은 R0로 이동시키고 레지스터를 되돌리며 반환을 수행하는 것으로 서브루틴을 종료한다.

2-5. 저장하는 주소

값을 불러오는 주소 또는 저장하는 주소로 인해 코드에 immediate값들이 다소 포함되어 있다. 이는 대부분 LDR주소나 STR주소를 불러오기 위함으로, 20*20 순회 시, 필요한 값들만 불러오고, 80*80 저장 시 올바른 위치에 저장하기 위함이다.

연산 시, 해당하는 행의 값과 다음 행의 값이 필요하기에 다음 행 값 호출을 위해 $+(20*4)$ 를 사용하며, 저장 시에는 다음 행에 저장을 위해 $+(80*4)$ 를 한 주소에 저장하게 된다. 또, 한 행을 완전히 처리한 후에는 다음 저장 시, 3개의 행을 건너뛰고 저장해야 하기에 $+(80*4*3)$ 의 수치를 적용한다. 각 레지스터가 저장하고 있는 현재 위치에서 더한 값을 사용하기에 각 값들은 위 수치에서 4~16의 차이가 발생할 수 있다.

3. Result

작성한 과제 코드의 예상 결과, 실행 결과 등을 figure와 함께 작성합니다. 어셈블리프로그래밍 수업에서는 muVision에 있는 debugging 기능을 활용하여 레지스터, 메모리의 값의 변화를 첨부합니다.

3-1. 메모리 로드 및 저장

Memory 2	
Address:	0x00000340
0x00000340:	00 00 80 3F 00
0x00000390:	00 00
0x000003E0:	00 00
0x00000430:	00 00
0x00000480:	00 00
0x000004D0:	00 00
0x00000520:	00 00
0x00000570:	00 00
0x000005C0:	00 00
Memory 1	
Address:	0x10000000
0x10000000:	00 00 80 3F 00 00 40 3F 00 00 00 3F 00 00 80 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10000050:	00 00
0x100000A0:	00 00
0x100000F0:	00 00
0x10000140:	00 00 40 3F 00 00 10 3F 00 00 C0 3E 00 00 40 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10000190:	00 00
0x100001E0:	00 00
0x10000230:	00 00
0x10000280:	00 00 00 3F 00 00 C0 3E 00 00 80 3E 00 00 00 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100002D0:	00 00
0x10000320:	00 00
0x10000370:	00 00
0x100003C0:	00 00 80 3E 00 00 40 3E 00 00 00 3E 00 00 80 3D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10000410:	00 00
0x10000460:	00 00

Figure 4. 20*20 순회 도중 메모리 상태 1

위 이미지는 20*20 순회의 첫 값에서의 메모리 상태를 확인한 결과이다. 메모리 주소의 50_{16} 차이는 십진수 기준 80 차이를 의미한다. 첫 번째 행의 1, 2열 값과 두 번째 행의 1, 2열 값이 로드되는 것을 확인할 수 있다. 또, 아래쪽의 0x10000000 메모리는 저장되는 위치로 위에서 불러온 데이터를 바탕으로 각 행의 처음 4열에 보간 결과가 들어가는 것을 확인할 수 있다. 140_{16} 차이는 십진수 기준 320 차이를 의미한다.

Memory 2	
Address:	0x00000340
0x00000340:	00 00 80 3F 00
0x00000390:	00 00
0x000003E0:	00 00
0x00000430:	00 00
0x00000480:	00 00
0x000004D0:	00 00
0x00000520:	00 00
0x00000570:	00 00
0x000005C0:	00 00
Memory 1	
Address:	0x10000000
0x10000000:	00 00 80 3F 00 00 40 3F 00 00 00 3F 00 00 80 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10000050:	00 00
0x100000A0:	00 00
0x100000F0:	00 00
0x10000140:	00 00 40 3F 00 00 10 3F 00 00 C0 3E 00 00 40 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10000190:	00 00
0x100001E0:	00 00
0x10000230:	00 00
0x10000280:	00 00 00 3F 00 00 C0 3E 00 00 80 3E 00 00 00 3E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100002D0:	00 00
0x10000320:	00 00
0x10000370:	00 00
0x100003C0:	00 00 80 3E 00 00 40 3E 00 00 00 3E 00 00 80 3D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10000410:	00 00

Figure 5 20*20 순회 도중 메모리 상태 2

위 이미지는 Figure 4의 상황에서 두 번째 값까지 연산한 결과이다. 1, 2번 행의 3번 열 값이 정상적으로 불러와졌으며, 2열의 값과 참조하여 연산된 후, 아래쪽 저장 메모리에 정상적으로 저장되었음을 확인할 수 있다.

3-2. 각 보간 값의 연산 과정

보간 값의 연산 과정에서 곱연산 및 합연산이 정상적으로 수행되는지 확인한다.

연산 예시로 사용되는 값은 0.txt의 일부분을 사용하였다. Figure 6에 기록된 부분의 연산 수행을 확인하였다.

Figure 7에서 연산 전 레지스터 상황을 확인할 수 있다. 각 값이 R2-R5에 입력된 상태임을 확인할 수 있다.

23	63.25	103.5	143.75	184
72.5	104.8125	137.125	169.4375	201.75
122	146.375	170.75	195.125	219.5
171.5	187.9375	204.375	220.8125	237.25
221	229.5	238	246.5	255
0x41B80000	0x427D0000	0x42CF0000	0x430FC000	0x43380000
0x42910000	0x42D1A000	0x43092000	0x43297000	0x4349C000
0x42F40000	0x43126000	0x432AC000	0x43432000	0x435B8000
0x432B8000	0x433BF000	0x434C6000	0x435CD000	0x436D4000
0x435D0000	0x43658000	0x436E0000	0x43768000	0x437F0000

Figure 9 올바르게 연산되었을 때의 예상 값

Figure 9 은 위 상황에서 올바르게 연산이 수행되었을 때의 예상 값이다.

Figure 7 상황은 위 설명과 더불어 $(1 - \frac{j}{4})V_1$ 이 수행되기 직전 상태로 $j = 0$ 이므로, multiply 서브루틴에 넘길 값인 $V_1 = 41B80000_{16}$ 과 $1 - \frac{4}{4} = 1 = 3F800000_{16}$ 이 저장되어 있는 것을 볼 수 있다.

Figure 8 은 연산 결과로 R0에 $41B80000_{16}$ 이 저장되었음을 확인할 수 있다.

같은 데이터의 다른 위치의 연산을 살펴보면, Figure 10의 상황을 볼 수 있다. 이는 $i=1, j=2$ 인 상황으로, Figure 9 기준으로는 146.375 값이 저장된 위치의 연산이다.

곱연산 직전의 레지스터로 $\frac{3}{4}V_1$ 연산을 위해 R1에는 $0.75 = 3F400000_{16}$ 가 저장된 것을 확인할 수 있다.

Current	
R0	0x418A0000
R1	0x3F400000
R2	0x41B80000

Figure 11 곱연산 후 결과

곱연산 후 결과는 $418A0000_{16}$ 으로 17.25가 저장된 것을 확인할 수 있다.

```
00 00 B8 41 00 00 38 43
00 00 5D 43 00 00 7F 43
```

Figure 6 보간 연산 예시

Current	
R0	0x41B80000
R1	0x3F800000
R2	0x41B80000
R3	0x43380000
R4	0x435D0000
R5	0x437F0000
R6	0x00000000
R7	0x00000000
R8	0x00000330
R9	0x00000458
R10	0x00000000
R11	0x00000000
R12	0x1000FA0
R13 (SP)	0xFFFFFDD0
R14 (LR)	0x00000040
R15 (PC)	0x000000F0
CPSR	0x80000003
SPSR	0x00000000

Figure 7 연산 전 레지스터 상황

Current	
R0	0x41B80000
R1	0x3F800000

Figure 8

Current	
R0	0x41B80000
R1	0x3F400000
R2	0x41B80000
R3	0x43380000
R4	0x435D0000
R5	0x437F0000
R6	0x00000008
R7	0x00000004
R8	0x00000330
R9	0x41B80000
R10	0x42DD0000
R11	0x41B80000
R12	0x10001224
R13 (SP)	0xFFFFFDD0
R14 (LR)	0x00000158
R15 (PC)	0x000000F0
CPSR	0x80000003
SPSR	0x00000000

Figure 10 연산 전 레지스터 상황



Figure 13



Figure 12

Figure 13은 $\frac{j}{4}V_2$ 연산 직전 Figure 12는 그 직후의 레지스터 상황이다. R0는 $\frac{j}{4} = 0.25$ 이고, R1은 $43380000_{16} = 184$ 이다. 연산 결과는 $42380000_{16} = 46$ 으로 정확히 연산되었음을 확인할 수 있다.



Figure 15

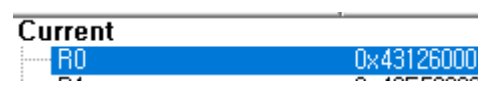


Figure 14

Figure 15 는 $(1 - \frac{i}{4})((1 - \frac{j}{4})V_1 + \frac{j}{4}V_2)$ 와 $\frac{i}{4}((1 - \frac{j}{4})V_3 + \frac{j}{4}V_4)$ 를 합하기 직전의 레지스터 상태이며, Figure 14는 그 직후 상태이다. $0.5 \times (0.75 \times 23 + 0.25 \times 184) = 31.625 = 41FD0000_{16}$ 을 확인할 수 있고, $0.5 \times (0.75 \times 221 + 0.25 \times 255) = 114.75 = 42E58000_{16}$ 을 확인할 수 있다. 결과는 $43126000_{16} = 146.375$ 로 정확히 반환되는 것을 확인할 수 있다.

3-3. 최종 결과



Figure 17 0.txt 결과


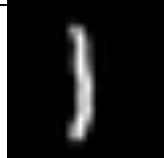



Figure 18

```
Please enter a number: 0
Start translating HEX data
Done!
Start calculating PSNR of
0: 32.57264451933952
Done.
All programs have finished
계속하려면 아무 키나 누르
```

Figure 16 PSNR 결과

위 Figure 18, Figure 17, Figure 16는 0.txt에 대한 결과로 Keil muVision과 프로젝트 과제 공지 시 제공된 hex to image compare 배치 파일을 실행한 결과이다. Little Endian으로 출력 시 배치 파일이 제대로 작동하지 않는 문제가 있으나, hex to image compare 배치 파일이 실행하는 파이썬 파일에서 IEEE to 754 Convert 함수를 수정하여 정상적으로 출력하도록 수정하였다.

입력	State	출력이미지	PSNR
0.txt	3157527		32.57264451933952
1.txt	2814162		35.84391560480219
2.txt	3123165		32.844608942307886

3.txt	3107604	3		32.815515507264095
4.txt	3007380	4		33.53466446919702
5.txt	3001234	5		33.942070520955916
6.txt	2989478	6		33.71456402736672
7.txt	3006745	7		33.843281895571636
8.txt	2989722	8		33.99691525248167
9.txt	2944536	9		33.9271455873932

Table 1 제공된 예시 출력 결과

위 표는 프로젝트 시작과 함께 제공된 예시 입력 데이터를 바탕으로 출력을 계산한 결과이다. Rounding을 수행하지 않았기에 PSNR값이 다소 낮음을 알 수 있고, 합연산 및 곱연산의 State 증가를 최적화하지 않아 다소 높은 State를 보임을 알 수 있다.

4. Discussion and Conclusion

4-1. 서브루틴의 진입과 반환에 대하여

서브루틴을 함수처럼 사용하는 것에 대해 어떤 방식으로 매개변수 전달과 반환을 수행할지 고민하였다. 레지스터에 저장하고 그 값을 그대로 사용하는 것을 고민하였으나, 전달할 값의 개수가 너무 많은 경우에 문제가 발생할 수 있다는 가능성을 고려하여 메모리에 값을 전달하는 방식도 고민해 보았다. 결과적으로 이번 프로젝트에서는 각 서브루틴이 큰 데이터를 처리하지 않고 국소적 값에 대해서만 연산을 수행하기에 해당 값들과 저장할 위치만 전달하였다. 이를 전달하기 위해서 큰 공간을 요하지 않기에 레지스터를 이용하여 변수의

전달과 반환을 수행하였다. 또, 해당 전달 과정과 반환 과정에 있어 매개변수의 전달 개수와 PUSH할 레지스터의 개수는 크게 중요하지 않으며, 반환하고자 하는 값의 개수에 따라 PUSH할 레지스터는 사용할 레지스터의 개수에서 반환하고자 하는 값의 개수만큼의 레지스터를 제하고 PUSH해야 함을 알 수 있었다.

서브루틴 작업 시에는 서브루틴에서 사용할 수 있는 최대한의 레지스터를 사용하고자 하였다. 메모리 오버헤드는 레지스터에 값을 주고받는 것보다 긴 시간을 요하는 것으로 알고 있다. 따라서, 프로젝트 진행 초반에는 최대한 메모리는 입출력값을 저장하는 것에만 사용하고자 하였다. 메모리 주소를 옮겨가며 이를 수행하고자 하였으나, 결국 IEEE - 754 Floating Point 연산이 레지스터를 너무 많이 사용하는 문제가 있어 이 해결을 위해 해당 연산을 수행할 때 오버헤드가 발생하게 되었다. 이 과정에서 메모리 주소를 옮기는 데 있어 Immediate Value를 최적화 하는 과정이나, 연산 과정에서 레지스터를 돌려쓰는 요령을 알 수 있었고, 이에 시도는 해보았지만 결국 레지스터 개수 부족으로 오버헤드를 발생시킬 수밖에 없었다.

4-2. 정수 연산과 IEEE - 754 연산의 정확도와 속도 차이

처음에는 값이 모두 정수로 주어진다는 사실을 알고, IEEE - 754로 표현된 값을 정수로 바꾸어 연산하여야 겠다고 생각하였으나, 프로젝트 의도에 부합하지는 않다고 생각하여 Github Repo에 Issue로 질문하였다. 정수로 바꾸어 연산하지 말라는 답변이 있었기에 IEEE - 754 방식으로 프로젝트를 진행하였으나, 해당 답변을 받기 이전에 정수로 변환하는 코드를 작성한 이력이 있어 해당 방식으로도 구현해본 결과 Rounding을 하지 않는다는 전제 하에 같은 PSNR값을 가지면서 더 빠르게 작동하는 프로그램을 작성할 수 있었다.

		0	1	2	3	4	5	6	7	8	9	AVG
IEEE 754	States	3157527	2814162	3123165	3107604	3007380	3001234	2989478	3006745	2989722	2944536	3014155
	PSNR	32.57264	35.87391	32.8446	32.81551	33.53466	33.94207	33.71456	33.84328	33.99491	33.92714	33.70633
Integer	States	360622	338171	356527	356865	349364	350014	349637	350079	347947	346907	350613.3
	PSNR	32.57264	35.87391	32.8446	32.81551	33.53466	33.94207	33.71456	33.84297	33.99691	33.96047	33.70983

Figure 19 정수 변환 후 연산한 결과의 State, PSNR

위 사진은 정수로 변환 후 연산을 진행한 뒤 다시 IEEE - 754로 변환하였을 때의 States 수를 기록한 것이다. 정수로 변환하는 과정에서 생각보다 많은 State를 증가시키지 않았으며, 정수 연산은 합, 곱연산에서 각각 State를 1, 2밖에 증가시키지 않으니 당연한 결과라고 생각한다. Rounding 연산을 수행하지 않는다는 가정이라면 IEEE - 754로 연산을 수행하였을 때는 Code-Size와 State면에서 좋지 못하다는 사실을 알 수 있었다.

4-3. IEEE - 754 연산의 최적화

프로젝트 초반에 Floating Point의 합연산과 곱연산 코드를 작성하고, 시간이 지난 뒤 Interpolation 코드를 작성하였는데, 과정 중 Arm Instruction 문서들을 살펴보다 이 명령어를 사용하였다면 더 적은 State를 사용했을 것이라 생각이 드는 명령어들이 여럿 있었다.

예시로, CLZ(Count Left Zero), MSB부터 0의 개수를 세는 명령이다. 이를 이용하면 더 빠르게 Normalize를 수행할 수 있어 크게는 약 100의 State를 아낄 수도 있었을 것이다.

MVN 명령의 경우에도 연산 처리 시 Negative로 바꾸기 위한 OER 연산의 다른 피연산자로 FFFFFFFF가 필요하였는데, 이를 MVN 0를 통해 쉽게 만들 수 있다. 메모리 호출이나, 다른 과정을 요하지 않는다.

외에도 ROTATE 명령 등을 활용하였다면 더 쉽게 구현하거나 더 빠르게 구현했을 작업들이 있었던 것으로 기억하고 있다. 이들을 이용해 State를 줄인다면 반복 횟수가 많은 특성 상 대략 10만의 State를 줄일 수 있지 않을까 생각한다.

4-4. 디버깅 도구

디버깅 과정에 있어 IEEE - 754 Converter를 계속해서 사용하는 것은 상당히 번거로운 과정이라 판단하였다. IEEE - 754로 표현된 Floating Point 값의 경우 16진수에서 10진수로 변환하는 과정은 상당히 번거로워 해당 도구를 사용할 수밖에 없었지만, 10진수에서 IEEE-754로 변환하는 경우, Excel에서 이를 수행하는 코드를 인터넷에서 찾을 수 있었다. 이를 바탕으로 Excel에서 보간을 수행하는 시트를 제작하고 그 값을 모두 IEEE - 754 방식으로 16진수 표현을 해주는 Table을 만들어 디버깅 과정을 수월하게 할 수 있었다. 디버깅 과정에서 다양한 도구를 도입하는 것으로 도구를 제작하는 시간보다 더 많은 시간을 절약할 수 있다는 사실을 알 수 있었다.

Reference

[IEEE - 754 Converter](#)

[About Excel IEEE - 754 Converting](#)

명예서약서 (Honor code)

“꿈을 가지십시오. 그리고 정열적이고 명예롭게 이루십시오.”

본인은 [어셈블리프로그램설계및실습] 수강에 있어 다음의 명예서약을 준수할 것을 서약합니다.

1. 기본 원칙

- 본인은 공학도로서 개인의 품위와 전문성을 지키며 행동하겠습니다.
- 본인은 안전, 건강, 공평성을 수호하는 책임감 있는 공학도가 되겠습니다.
- 본인은 우리대학의 명예를 지키고 신뢰받는 구성원이 되겠습니다.

2. 학업 윤리

- 모든 과제와 시험에서 정직하고 성실한 태도로 임하겠습니다.
- 타인의 지적 재산을 존중하고, 표절을 하지 않겠습니다.
- 모든 과제는 처음부터 직접 작성하며, 타인의 결과물을 무단으로 사용하지 않겠습니다.
- 과제 수행 중 받은 도움이나 협력 사항을 보고서에 명시적으로 기재하겠습니다.

3. 서약 이행

본인은 위 명예서약의 모든 내용을 이해하였으며, 이를 성실히 이행할 것을 서약합니다.

학번 | 2023202027 | 이름 | 정정윤 | 서명 | 정정윤