어셈블리프로그램 설계및실습 보고서

Term project

학 과: 컴퓨터정보공학부

담당교수: 이형근 교수님

실습분반: 화요일 6, 7

학 번: 2019202050

성 명: 이강현

제 출 일: 2022.12.3(토)

1. Introduction

   이번 프로젝트에서는 부동소수점으로 이루어진 이산신호와 부동소수점으로 이루어진 필터와의 convolution을 진행하고 이에 대한 결과중 조건에 맞는 결과와 인덱스만을 메모리에 저장하는 프로그램을 구현한다.

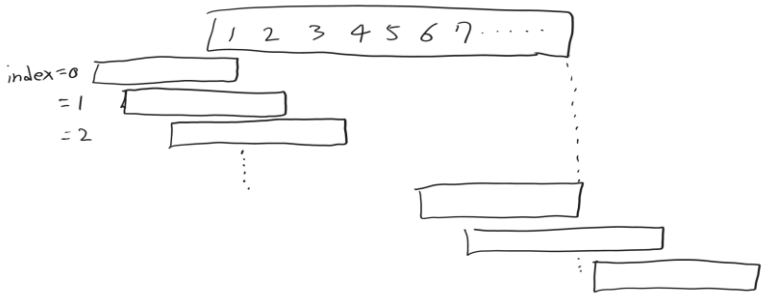
2. Background

   해당 프로젝트를 구현하기 위해서는 우선 convolution에 대해 알아야 한다. 사용하는 convolution은 1D-convolution이며 예를 들어 1,2,3,4,5라는 이산신호와 (1,2)라는 필터를 사용하여 둘을 convolution한다면 필터 혹은 이산신호를 뒤집은 뒤 인덱스를 바꿔가며 곱한다. 인덱스가 0이라면 2*0+1*1 1이라면 2*1+1*2, 2이라면 2*2+1*3, 3이라면 2*3+1*4 4라면 2*4+1*5 5라면 2*5+1*0으로 연산한다. 필터를 뒤집어서 연산하는 이유는 왼쪽에 있는 값일 수록 먼저 오른쪽으로 진행되서 입력되기 때문이고 이러한 합성곱은 시간 불변성 때문에 생기는 딜레이를 고려하여 필터와 값들을 곱해주는 것이고 이를 선형성에 기인하여 더해주어 값을 구하는 방식이다.

   또한 부동소수점 표현에 대해 알아야한다. 이번 프로젝트에서는 32비트 값들을 서로 곱해주는 연산을 진행하므로 32비트의 값이 부호비트 1비트, 지수를 표현하는 비트 8비트, 1.xxx* 2의 지수승에서 xxx에 해당하는 실질적인 값들을 나타내는 mantissa 23비트로 이루어진다는 것을 알아야 한다.

3. Algorithm

   우선 프로젝트 구현은 아래와 같은 방식으로 구현하였다. 따라서 인덱스는 480이상이 나온다.



이번 프로젝트에서는 서브루틴 함수를 많이 사용하였다. 먼저 r0에 이산신호의 주소를 저장해두고 r1에 필터의 주소를 저장해 두었다. r2에는 결과값을 저장할 곳의 주소를 저장해두고 r3는 이산신호의 인덱스, r4는 필터의 인덱스를 저장해두는 것으로 5개의 regiser들은 연산이 진행되는 동안 다른 값의 개입없이 고정적으로 변하는 register이다.

r6부터 r12까지는 덧셈연산과 곱셈연산등을 진행할 때 필요한 register들이 많기에 중복해서 사용한다. R13은 최종결과값을 저장하는 register로 사용하였고 따라서 곱셈의 결과값이자 덧셈의 피연산자로서 사용된다.

우선 프로그램이 시작되면 r3를 1920(480*4)와 비교하여 이보다 작지 않으면 연산을 시작하지 않는다. 그 이유는 0부터 시작하여 총 480개의 데이터가 있으므로 479의 인덱스까지 값이 존재한다. 데이터는 4바이트단위로 저장되어 있기 때문에 주소 값기준 1920에는 데이터가 없다. 따라서 곱셈시 480이전의 값만 연산을 진행한다. 그후 데이터를 불러와 r5,r6에 저장하고 multiplier라는 서브루틴을 실행한다. Multiplier는 곱셈을 진행하는 함수로 sign bit,exponent,mantissa를 각자 분리하여 저장하고 signbit가 같다면 최종결과의 signbit는 0 아니라면 1을 저장한다. 그후 exponent를 서로 더해주고 127을 빼준 후 저장한다. 마지막으로 핵심은 mantissa의 곱셈이다. 23bit와 23bit의 곱셈이므로 32bit이상의 결과값을 얻게된다. 게다가 부호의 영향을 받지 않기 때문에 UMULL을 사용하여 상위비트와 하위비트를 나눠 저장한다. 둘을 곱셈할 경우에 1.xx와 1.xx형식에서 곱셈을 하므로 1.xxx꼴로 나오는 경우가 있고 10.xxxx혹은 11.xxxx를 뽑아내는 경우가 있을 것이다. 1.5와 1.5를 곱한다면 2.25가 나오는 것을 생각하면 된다.

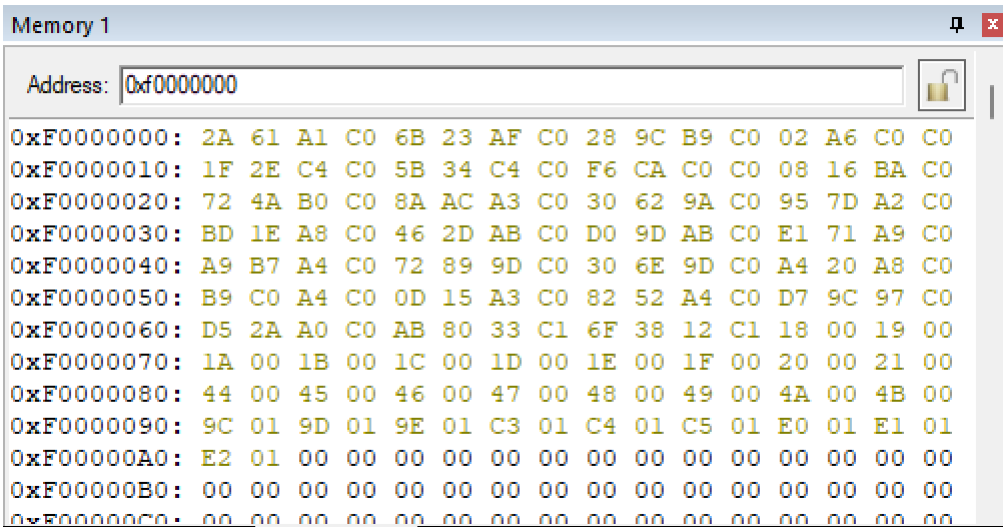
그렇다면 1.xxx꼴로 나오는 것은 최종 결과 값이 47비트로 10.xxx나 11.xxx가 나온다면 48비트를 최종 결과 값으로 얻게 되는 것이다. 따라서 하위비트 32비트를 제하고 상위비트가 15비트인지 16비트인지로 경우를 나누어 15비트라면 하위비트의 상위비트 8비트를 가져오고 16비트라면 상위비트 9비트를 가져와야 오차를 최대한 줄일 수 있다. 이렇게 연산한 후 자릿수를 맞춰주면 곱셈연산이 완료된다. 곱셈연산을 하게 되면 바로 덧셈연산으

로 흘러가는데 처음에는 덧셈연산시 비어있는 r13의 값 0과 연산하므로 해당조건을 adder 함수에 넣어주어 바로 연산을 마무리하게끔 하였다. 이로서 state를 아낄 수 있다.

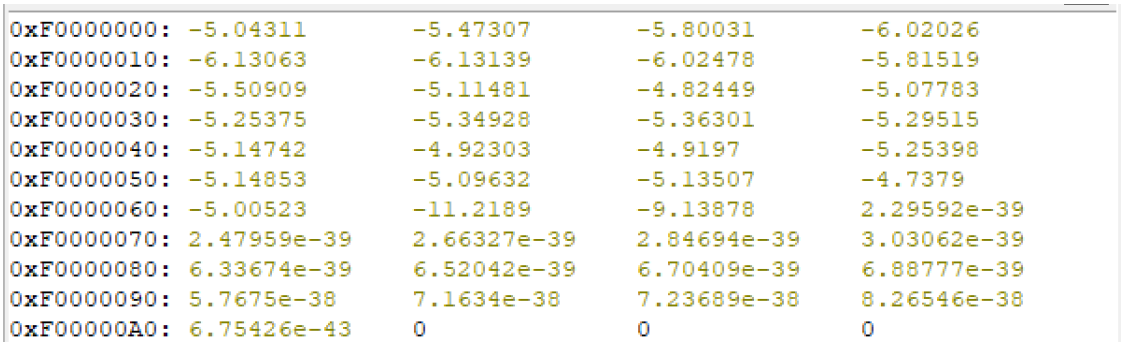
덧셈은 이전에 구현했던 adder를 기준으로 진행했으며 register의 사용방법만 바꾸었다. adder또한 사인비트,지수비트,mantissa를 나누어 저장하고 signbit가 같을 때와 다를 때로 나누고 지수비트가 다를 때 shift를 통해 큰 값에 맞춰주고 그 후 위치가 달라진 mantissa를 연산한 후 normalization을 반복하여 자리를 맞춰준 후 연산을 위해 사용했던 1을 감해준 뒤 결과를 구하면 된다.

이렇게 곱셈과 덧셈 연산이 마무리되면 다시금 서브루틴을 실행했던 장소로 돌아가 필터 인덱스를 +1해주고 데이터 인덱스는 -1을 해준 뒤 서로가 가리키는 값을 연산한다. 이는 위에서 설명한 합성곱의 연산 방식 때문에 더해주는 값이 다름을 인식한다. 그 후 데이터 인덱스는 음수가 되면 negdataindex함수로 이동하여 데이터 인덱스는 이전연산보다 4바이트 이동한 값을 가리키게 끔하고 필터 인덱스는 다시 첫 필터 값을 가리키게끔 초기화한다. 이렇게 초기화 된다면 현재 인덱스에서의 합성곱 연산을 마무마무리한 것이 endloop함수로 이동하여 결과값을 저장할 지 하지 않을지를 결정한다. 이를 결정하는 함수는 Result함수이다. Result함수에서는 -4.7이라는 기준 값과 사인비트, 지수비트, mantissa값을 순서대로 비교하고 비교 후 조건이 충족되면 r2의 주소에 값을 저장한다. r2의 2000만큼의 주소 뒤에는 해당 인덱스를 저장한다. 모든 인덱스의 값이 저장되더라도 1920바이트이상의 공간이 사용될 수 없다. 따라서 2000이후만큼에 인덱스를 순차적으로 저장해둔 후 indexalign 함수를 통해 실제 결과값 뒤에 붙이는 작업을 실행할 것이다. 결과값이 모두 저장되고 모든 연산이 마무리되면 위에서 언급한대로 indexalign함수가 실행된다. Indexalign 함수가 실행되기 전 endloop에서는 결과값들의 개수를 주소 값들의 차로 계산하여 register에 저장해두고 이 개수만큼 indexalign을 반복하여 결과값 바로 뒤에 STRH명령어를 통해 word단위로 저장한다. 이렇게 모든 흐름이 마무리된다

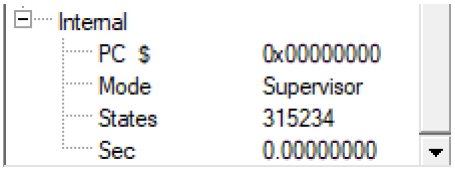
4. Performance & Result



최종결과 사진이다. 0xF0000060의 18부터 인덱스 값이다. 그전의 결과값은 double word(32bit)단위로 저장되었지만 인덱스 값은 word(16bit) 단위로 저장됨을 알 수 있다. 총 27개의 값이 저장되었고



위는 float 형식으로 출력한 값이다.



315234의 state가 실행되었다.

5. Consideration

이번 프로젝트에서는 부동소수점 연산과 서브루틴, 메모리에 데이터 저장 및 불러오기등 그동안 실습했던 모든 내용들을 모두 이해하고 사용해야 했다. 특히 부동소수점에서 bit단위로 연산하고 자릿수를 맞춰주고 하는 부분이 어려웠다. 또한 합성곱이라는 개념이 낯설어서 이를 이해하는데도 시

간이 많이 걸렸다. 인덱스가 0부터 시작하나 필터를 뒤집어서 연산하므로 0과 곱해지는 경우도 생각하여 구현하였기 때문에 단순한 곱셈연산이지만 흐름을 파악하고 일반화하여 함수로 구현할 방법을 생각하는데 시간을 많이 사용한 것 같다. 또한 연산을 register에서만 진행해야 하는 점이 많이 어려웠다. 기존의 c++같은 언어는 변수를 원하는 만큼 생성할 수 있어 편했으나 연산이 진행되는 부분을 register로 제한해야 한다는 점이 어려웠다. 고정적으로 변하는 값을 5개를 사용했고 덧셈연산과 곱셈연산을 모두 해야하지만 register가 한정적이라 덮어씌워서 사용했고 이러한 방법이 실제 연산의 결과나 연산중 데이터에 영향을 끼치지 않아야 했으므로 서브루틴을 진행하는 흐름 또한 중요했다. 따라서 stack memory까지 사용하였다. 또한 인덱스를 저장하는 부분도 힘들었다. 조건에 맞는 결과가 총 몇 개인지 연산이 모두 진행되기 전에는 알 수 없기에 결과를 저장하고 인덱스를 저장해야하는 이러한 흐름에서는 인덱스를 메모리가 아닌 곳에 저장해 둘 수가 없었다. 따라서 우선 저장해둔 후 끌어오는 과정이 필요했다. 프로젝트를 진행하면서 실습했던 기본적인 개념을 다시 확인하는 경우도 많았고 까먹었던 내용들을 다시 학습할 수 있는 좋은 기회였던 것 같다.

6. Reference

이형근/어셈블리프로그램 설계 및 실습/광운대학교(컴퓨터정보공학부)/2022