어셈블리 프로그램

설계 및 실습

Term Project 결과 보고서

* Matrix Multiplication -



학 과: 컴퓨터공학과

담당교수: 이준환 교수님

학 번: 2012722028

성 명: 장 한 별

1. **Introduction**

해당 project는 Floating Point 로 이루어진 임의의 Data 1800 개를 각 순서에 맞춰 곱셈 연산과 덧셈 연산 후 30 X 30 정방행렬 꼴로 Memory 에 Store 하는 어셈블리 프로그램이다. 이때 사용하는 연산은 Floating Point Multiplication, Floating Point Addition 이 있는데 Multiplication에선 Booth Multiplication(Radix-2) 방식으로 구현되었다. Booth Multiplication 과 두 정방행렬의 곱셈 연산에 대한 자세한 내용은 뒤에서 다루기로 한다.

**▶ Project schedule**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 주차 | 일 정 | 추진 일정 | | |
| 1 | 2 | 3 |
| 1 | Floating Point Multiplication 구현 (Booth Multiplication 방식) |  |  |  |
| 2 | Floating Point Addition 및  30 X 30 정방행렬 구현 |  |  |  |
| 3 | 최종 검증 및  결과보고서 작성 |  |  |  |

**►** 1주차 ~ 2주차 : Floating Point Multiplication 을 Booth Multiplication 방식으로 구현한다. 이때 Booth Algorithm 중 Radix-2 booth multiplication 방식으로 구현했다.

**►** 2주차 ~ 3주차 : Floating Point 과제 중 Addition 부분을 대폭 수정했다. Exponent를 adjust 하는 부분에 더 많은 예시를 입력시켜 검증을 보다 확실하게 수행했다.

**►** 3주차 : 30 X 30 정방행렬 연산을 구현하고 정확한 주소에 Store 가 되는지 확인했다.

**►** 17. 11. 27 : 최종 보고서 제출

1. **Project Specification**

**► Overall**

: Matrix\_data 의 주소를 가져와 1번째 원소와 901 번째 원소를 Load 시킨 후 각 주소값을 4byte, 120 byte씩 증가시킨다. 이렇게 Load 시킨 원소들을 Mmul\_data 에서 NaN인지 아닌지, 0인지 아닌지 확인 후 Sign bit, Exponent, Mantissa 를 각각 추출시켜 Floating Point Multiplication 연산을 한다. 이때 Multiply 연산 종류 중 Booth Multiplication 방식을 사용했다. 그 중에서도 Radix-2 방법을 사용했다. 이 방식은 Mmul\_data에서 다루기로 한다. 그 후 Multiply 연산 후 나온 결과값과 기존 Madd\_data 의 결과값을 더하는 Madd\_data 연산을 한다. Madd\_data 역시 Sign bit, Exponent, Mantissa 를 추출시켜 Floating Point Addition 연산을 한다.

이렇게 나온 결과값들로 30 X 30 정방행렬을 구성한다. 행과 열의 숫자가 비교적 적은 2 X 2 정방행렬의 곱으로 알아보도록 하자. 2 X 2 정방행렬의 곱은 다음과 같다.

**►►** 정방행렬의 곱셈

- 2 X 2 정방행렬 A, 와 2 X 2 정방행렬 B 의 곱셈 연산 후 나오는 2 X 2 정방행렬을 C 라 할 때, 곱셈 연산 시 행렬 C의 각 원소들은 다음과 같이 이루어진다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **A11** | **A12** | **X** | **B11** | **B12** | | **A21** | **A22** | **B21** | **B22** |  |  |  |  | | --- | --- | --- | | **=** | **A11B11 +A12B21** | **A11B12 + A12B22** | | **A21B11 + A22B21** | **A21B12 + A22B22** | |
| 두 2X2 정방행렬 의 곱셈 연산 |

**► Mmul\_data**

: Load 된 2개의 value 들을 곱하는 영역이다. 이때 각 value 들은 Floating point로 이루어져 있어, 일반적인 곱셈 방식으로 곱셈 연산을 수행하면 안된다.

먼저, Load 된 value 들을 각각 Sign bit, Exponent, Mantissa 를 추출 후 결과값의 Sign bit 은 두 Sign bit 이 같을 때는 0, 다를 때는 1, 즉 XOR 연산을 통해 구할 수 있다. 결과값의 Exponent 는 두 Exponent를 더한 후 127을 빼주면 되는데 Mantissa 의 값에 따라 Adjust 될 수 있다. 결과값의 Mantissa 는 Booth Multiplication 방식으로 곱해야 하는데 Booth Multiplication(Radix-2) 방식은 다음과 같다.

**►►**Booth multiplication(Radix-2)

|  |  |  |  |
| --- | --- | --- | --- |
| <Rule of booth multiplication> | | | |
| xi | xi-1 | Operation | Description |
| 0 | 0 | Shift only | String of zeros |
| 0 | 1 | Add and shift | End of a string of ones |
| 1 | 0 | Subtract and shift | Beginning of a string of ones |
| 1 | 1 | Shift only | String of ones |
| 위 그림은 booth multiplication을 하기 위한 기본적인 규칙이다.  Multiplication은 Multiplicand(피승수) Multiplier(승수)를 곱하는 연산이다. 이때 booth multiplication의 방법은 두 개의 signed 2진 숫자를 2의보수의 원리를 이용하여 곱셈을 진행한다.  곱셈 방법은 Multiplier의 xi 와 xi-1을 1개씩 늘려가면서 비트 수만큼 비교를 하는 것이다.  자세히 말하자면, 일단 처음 Multiplier의 LSB(least significant bit)의 오른쪽에는 0비트가 있다고 가정한다. xi 와 xi-1 를 비교할 때(이때, xi와 xi-1을 비교하는 숫자는 multiplier이다), xi 가 0 이고 xi-1이 0 이면 오른쪽으로 1bit shift만한다. 이때, 모든 shift는 RSB이다. 그리고, xi 가 이고 xi-1이 1이면 add 후 오른쪽으로 1bit shift한다. 이때, add하는 숫자는 multiplicand이다. xi가 1이고, xi-1이 0이면, Subtract 후에 shift한다. 이때 subtract하는 숫자는 multiplicand이다.  subtract하는 방법은 2의 보수를 취한 뒤에 add를 하는 방식으로 하면 되겠다. 이렇게 진행하다 보면, 비트 수만큼 연산을 하게 되고 결과값을 도출할 수 있게 된다. Radix-2의 방법은 1이 연속적으로 많으면 좋은 곱셈 방식이지만, 0101 이런 식처럼 0과 1이 계속 교차되어 표현된 숫자의 곱셈연산에는 좋지 않을 수 있다. 따라서 이런 문제점을 보완하여 3bit씩 쪼개서 곱셈 연산을 진행하는 방식인 radix-4, radix-8 방식이 존재한다. | | | |

|  |  |  |
| --- | --- | --- |
| <Course of Radix-2 Booth Multiplication> | | |
| A  X  Y | 0101  0011(0)  0101’ | 5(Multiplicand) |
| 3(Multiplier) |
|  |
| Subtract A  Shift | 1011(inversion+1)  11011(RSB) |  |
|  |
| Shift | 111011(RSB) |  |
| ADD A  Shift | + 0101  001111  0001111 |  |
|  |
|  |
| Subtract A  Shift | 1011(inversion+1) |  |
| 1100111 |  |
| 11100111 |  |
| result | 11100111(2) | 15 |
| 위 과정은 booth multiplication의 radix-2 곱셈 과정을 나타낸 식이다.  위의 표에 나와있는 것처럼 xi 와 xi-1 를 비교할 때(이때, xi와 xi-1을 비교하는 숫자는 multiplier이다), xi 가 0 이고 xi-1이 0 이면 오른쪽으로 1bit shift만한다. 이때, 모든 shift는 RSB이다. 그리고, xi 가 이고 xi-1이 1이면 add 후 오른쪽으로 1bit shift한다. 이때, add하는 숫자는 multiplicand(5)이다. xi가 1이고, xi-1이 0이면, Subtract 후에 shift한다. 이때 subtract하는 숫자는 multiplicand(5)이다. subtract하는 방법은 2의보수를 취한 뒤에 add를 하는 하면 된다. | | |

따라서, 두 Floating Point Data Value1, Value2 를 곱셈 연산(Valueresult)은

Sresult = S1 ^ S2

Eresult = E1 +E2 -127

Mresult = M1 과 M2 의 Booth multiplication 한 결과.

이러한 연산으로 수행할 수 있다. Floating Point Multiplcation 부분은 Algorithm 에서 Flow char 와 Pseudo code를 작성하여 더 자세하게 알아보도록 한다.

**► Madd\_data**

: Mmul\_data 에서 나온 결과값과 이전 Madd\_data 의 결과값을 더하는 영역이다. 이때 두 value 들은 Floating point로 이루어져 있어, 일반적인 덧셈 방식으로 곱셈 연산을 수행하면 안된다.

각 두 value 에서 Sign bit, Exponent, Mantissa 들을 추출한다. 그 후 Exponent 를 비교후 Exponent 가 큰 값을 기준으로 Exponent 보다 얼마나 작은지 계산 후 그 만큼 Mantissa 를 shift 시킨다. 그 후 Mantissa를 더하고 이때 Carry 를 확인하여 Exponent 를 adjust 시킨다. sign bit 의 결과를 확인 후 Sign bit, Exponent, Mantissa 를 합쳐 최종 결과값을 내보낸다. Floating Point Addtion 부분은 Algorithm 에서 Flow chart 와 Pseudo code 를 작성하여 더 자세하게 알아보도록 한다.

1. **Algorithm**

**▶ Overall Project**

|  |
| --- |
| Flow chart |
|  |
| 위 그림은 이번 project 의 Flow Chart 이다. 30 X 30 정방행렬을 구성하기 위해 Count 가 증가한 값이 30에서 다음 Loop로 넘어가는 것이 매우 중요하다. |

|  |
| --- |
| Pseudo code |
| {  Load r4 = address of Value 1  Load r12 = address of Value 2  For( int I =1 ; I < 31 ; i++)  {  For( int J = 1; j < 31 ; j++)  {  For( int k = 1; k < 31 ; k++)  {  Load r1 from r4, r4 + = 4  Load r2 from r12, r12 + = 120  R3 = r1 \* r2 // Floating Point Multiplication  R9 = r9 + r3 // Floating Point Addition  }    Store r9 into r7, r7 += 4  Load r12 = address of Value2  R12 = r12 + 120 \* j  }    Load r4 = address of Value1  R4 = r4 + 120 \* I  Load r12 = address of Value2  }  } |
| 위 그림은 이번 project 의 Pseudo code 이다.  이 Pseudo code 에선 Floating Point Multiplication 부분과 Floating point Addition 부분은 간단하게 MUL, ADD instruction으로 표현했지만 실제 어떻게 곱셈 연산과 덧셈 연산이 이루어지는지 아래에서 확인하도록 한다. |

**▶ Floating Point Multiplication**

|  |
| --- |
| Flow chart |
|  |
| 위 그림은 이번 project 중 Floating Point Multiplication 부분의 Flow Chart 이다. |

|  |  |
| --- | --- |
| Pseudo code | |
| {  Load r1 = value 1  Load r2 = value 2  R3 = r1  Shift r3 to left 1, Shift r3 to right 24  If ( r3 = 255)  {  R4 = r1  Shift r4 to left 9, Shift r4 to right 9  If (r4 = 0)  Continue  Else  Call function Result  }    R3 = r2  Shift r3 to left 1, Shift r3 to right 24  If ( r3 = 255)  {  R4 = r1  Shift r4 to left 9, Shift r4 to right 9  If (r4 = 0)  Continue  Else  Call function Result  }  R3 = r1  Shift r3 to left 1, shift to right 1  If (r3=0)  Call function Result  R3 = r2  Shift r3 to left 1, shift to right 1  If (r3=0)  Call function Result    R3 = shift r1 to right 31  R4 = shift r2 to right 31  If ( r3==r4)  R3 = 0  Else  R3 = 1  R3 = shift r3 to left 31  R5 = shift r1 to left 1  R5 = shift r5 to right 24  If (r5 = 255)  {  R5 = shift r5 to left 23  Call function Result  }    R6 = shift r2 to left 1  R6 = shift r6 to right 24  If (r6 = 255)  {  R6 = shift r6 to left 23  Call function Result  } | R5 += r6  R5 -= 127    Make r7 mantissa form  Make r8 mantissa form  While (1)  {  R4 = LSB 2 bits  If (r4 = 2\_00)  Shift r9  continue  else If (r4 = 2\_01)  R9 += r7  Shift r9  continue  else If (r4 = 2\_10)  R9 -=r7  Shift r9  continue  else If (r4 = 2\_11)  Shift r9  continue  If ( MSB )  Break  }  R4 = shift r9 to right 23  If (r4 = 2\_00)  R5 –= 0x800000  Make R9 fraction  If (r4 = 2\_01)  Make r9 fraction  If (r4 = 2\_10)  R5 += 800000  If (r4 = 2\_11)  R5 += 800000  Make r9 fraction  R9 += 400000  }  Result (a, b)  {  R3 += a  R3 += b  Return r3  } |
| 위 그림은 이번 project 중 Floating Point Multiplication 부분의 Pseudo code 이다. | |

**▶ Floating Point Addition**

|  |
| --- |
| Flow chart |
|  |
| 위 그림은 이번 project 중 Floating Point Addition 부분의 Flow Chart 이다. |

|  |  |
| --- | --- |
| Pseudo code | |
| {  Load r1 = value 1  Load r2 = value 2  If( r1==0)  R10 = r2  Call function result  Else if( r2 ==0)  R10 = r1  Call function result    R3 = S of value 1  R4 = S of value 2  R5 = E of value 1  R6 = E of value 2  R7 = M of value 1  R8 = M of value2  If( r5 > r6)  R9 = r5- r6  R8 \*= r9  R11 = r5  Else  R9 = r6 – r5  R7 \*= r9  R11 = r6  If ( r3 == r4)  {  R12 = r7 + r8  If( r9 == 0x1000000)  {  Shift r12 to right 1  R11 += 1  }  R12 -= 0x800000  Shift r11  Shift r3  R10 = r3 + r11  R10 += r12  Call function result  } | Else  {  If (r5>r6)  R12 = r7 – r8  Else if ( r5<r6)  R12 = r8 - r7  R3 = r4  Else  {  If (r7>r8)  R12 = r7 - r8  Else if ( r7<r8)  R12 = r8 – r7  R3 = r4  Else  R10 = 0  Call function result  }  R0 =0  while (r9 != 0x800000)  {  Shift 12 to left 1  R0 += 1  }    R11 -= r0  Make r12 fraction  Shift r11  Shift r3  Call function result  }  }  Result (a, b)  {  R10 += a  R10 += b  Return 10  } |
| 위 그림은 이번 project 중 Floating Point Addition 부분의 Pseudo code 이다. | |

1. **Performance & Result**

**▶ Result of the test\_data\_1**

**► Result**

|  |
| --- |
| **Memory** |
| C:\Users\장한별 (Stanley)\Desktop\me1-1.JPG  C:\Users\장한별 (Stanley)\Desktop\me1-2.JPG |
| 위 그림은 test\_data1 의 결과 Memory에 store 된 900 개의 data들이다. 이때 첫 주소값은 0x60000000 이고 900개의 data 가 모두 store 된 후 0x6000E10 에 아무런 data가 store 되지 않았음을 확인할 수 있다. 이 900개 data는 Mmul\_data 와 Madd\_data 에서 연산을 수행 후 30 X 30 정방행렬을 구성한다. 올바르게 출력되었음을 확인할 수 있다. |

**► Performance**

|  |  |
| --- | --- |
| **State** | **Code Size** |
| C:\Users\장한별 (Stanley)\Desktop\state1.JPG | C:\Users\장한별 (Stanley)\Desktop\size1.JPG |
| 위 그림을 확인해보면 States 는 13,554,944, Code size는 8253 bytes 임을 확인할 수 있다. Score 는 Code size \* State2 으로 이루어지므로 Score 는 1,516,193,654,469,558,272 이다 | |

**▶ Result of the test\_data\_2**

**► Result**

|  |
| --- |
| **Memory** |
| C:\Users\장한별 (Stanley)\Desktop\me2-1.JPG  C:\Users\장한별 (Stanley)\Desktop\me2-2.JPG |
| 위 그림은 test\_data2 의 결과 Memory에 store 된 900 개의 data들이다. 이때 첫 주소값은 0x60000000 이고 900개의 data 가 모두 store 된 후 0x6000E10 에 아무런 data가 store 되지 않았음을 확인할 수 있다. 이 900개 data는 Mmul\_data 와 Madd\_data 에서 연산을 수행 후 30 X 30 정방행렬을 구성한다. 올바르게 출력되었음을 확인할 수 있다. |

**► Performance**

|  |  |
| --- | --- |
| **State** | **Code Size** |
| C:\Users\장한별 (Stanley)\Desktop\state2.JPG | C:\Users\장한별 (Stanley)\Desktop\size2.JPG |
| 위 그림을 확인해보면 States 는 13,424,423, Code size는 8256 bytes 임을 확인할 수 있다. Score 는 Code size \* State2 으로 이루어지므로 Score 는1,487,856,137,081,461,824 이다 | |

**▶ Result of the test\_data\_3**

**► Result**

|  |
| --- |
| **Memory** |
| C:\Users\장한별 (Stanley)\Desktop\me2-1.JPG  C:\Users\장한별 (Stanley)\Desktop\me2-2.JPG |
| 위 그림은 test\_data3 의 결과 Memory에 store 된 900 개의 data들이다. 이때 첫 주소값은 0x60000000 이고 900개의 data 가 모두 store 된 후 0x6000E10 에 아무런 data가 store 되지 않았음을 확인할 수 있다. 이 900개 data는 Mmul\_data 와 Madd\_data 에서 연산을 수행 후 30 X 30 정방행렬을 구성한다. 올바르게 출력되었음을 확인할 수 있다. |

**► Performance**

|  |  |
| --- | --- |
| **State** | **Code Size** |
| C:\Users\장한별 (Stanley)\Desktop\state2.JPG | C:\Users\장한별 (Stanley)\Desktop\size2.JPG |
| 위 그림을 확인해보면 States 는 13,424,423, Code size는 8256 bytes 임을 확인할 수 있다. Score 는 Code size \* State2 으로 이루어지므로 Score 는1,487,856,137,081,461,824 이다 | |

1. **Conclusion**

이번 Project 는 Floating Point 로 이루어진 data 1800 개를 900개씩 두 개의 30 X 30 정방행렬꼴을 만든 후 그 두개의 행렬을 곱셈 연산을 하는 Project 였다. 처음 Project 제안서를 받았을 때는 머릿속으로 어떻게 구현 해야겠다는 알고리즘이 떠올라 난이도가 쉽다고 생각했었다. 하지만 직접 Assembly 언어로 코딩을 해보니 쉽게 처리되지않는 부분이 한 두 곳이 아니었다.

먼저, Floating Point Addition 하는 부분은 이미 인터넷강의와 실습 시간에 익혀 두고, 과제를 통해 Project 에 대한 예습을 했다고 생각했지만, 고려해주지 않은 부분이 너무 많아 다시 처음부터 공부하고 숙지한 후 코딩해야 했다. data 를 load 후 Sign bit, Exponent, Mantissa 를 각각 추출 해 주는 부분에선, Fraction 값에 1을 더한 값이 Mantissa 꼴을 구성한다는 점을 놓친 점이 가장 큰 실수였다. 그 후 Exponent 를 비교하여 Mantissa 를 그 차이만큼 shift 시켜주는 부분은 무난하게 넘어갔으나, Mantissa 를 Normalize 후 Exponent Adjust 하는 부분에서 많은 노력을 기울였다. 특히 두 Value 가 보수 관계인, 즉, Sign bit 을 제외한 모든 bits 가 동일 한 경우 Mantissa 를 Normalize 하고 Exponent를 Adjust 하는 부분을 제대로 구현하지 못해 Addition을 모두 구현했다고 생각했지만, 행렬 원소를 구성하는 부분에서 연속으로 30번 더해줄 때 잘못된 것을 확인하여, 처음 Addition 값부터 하나하나 디버깅 하면서 발견해 매우 힘들었다.

그 후 Floating Point 에 대한 개념이 잡히고 어느정도 숙달되어 Multiplication 을 구현했다. Floating Point Multiplication 에선 Sign bit과 Exponent 는 쉽게 구할 수 있었지만, 이번 Project 의 핵심이라 느껴지는 Mantissa의 Booth Multiplication 부분에 정말 많은 노력을 기울였다. 제대로 구현했다고 생각했지만 2번째 Mantissa 에 뒤에 0이 있다고 생각하고 구현하는 부분을 바로 생각하지 못 해 다시 Booth Multiplication 을 공부했고, MSB 가 1 이면 음수로 처리되어 1보다 앞 bit 에 0을 추가해 구현했다. 많은 시간을 투자한 만큼 어떤 Data 가 들어와도 모두 처리되는 것을 확인했을 때 크나큰 희열을 느꼈다.

마지막으로 30 X 30 Matrix 의 곱을 구현 후 Floating Point 를 Multiply 하는 영역과 Add 하는 영역을 합쳤다. 바로 처리 될 줄 알았지만 각 Register 들이 중복으로 사용되어 원하는 결과값을 바로 얻을 수 없었다. 그래서 그렇게 중복되어 사용되는 Register 들과 계속 바뀌어야 하는 Register 들을 구분하고, Stack에 쌓아 기존 data를 따로 저장시켰고, 원하는 결과값을 얻을 수 있었다. 그렇지만 이 과정에서 R/W base 를 계속해서 변경해줘야했고, inifile 에 참조할 수 있는 주소들을 계속해서 수정해야 했다. 그 후 test\_data 의 결과값과 비교해주어 최종결과값이 올바르게 Memory에 Store 되었음을 확인함으로 이번 Project를 마무리 지었다.

이번 프로젝트를 통해 Assembly 언어와 더욱 친숙해진 것 같고, Hardware 로 가는 첫 번째 발걸음을 내딛은 것 같아 뿌듯하다.

**►** Reference

강의자료

이준환 / 어셈블리 프로그램 설계 및 실습 / 광운대학교 (컴퓨터공학과)/ 2017년