어셈블리 프로그래밍 설계 및 실습

실험제목: Floating-Point

실험일자: 2017년 10월 26일 (목)

제출일자: 2017년 11월 1일 (수)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 화 5, 목 6, 7

학 번: 2012722028

성 명: 장 한 별

1. 제목 및 목적 (3%)
   1. 제목

Floating - Point

* 1. 목적

Floating point의 개념을 이해하고, 실수를 Floating point로 표현하는 방법에 대해서 익히도록 한다. 또한 Floating point간의 덧셈 연산의 방법에 대해 이해하고, 이를 어셈블리 언어를 통해 구현 할 수 있도록 하는 데에 목적이 있다.

1. 설계 (Design) (50%)
   1. Pseudo code

|  |
| --- |
| {  R0 = value1, r1 = value2  If value = 0, value \*2 // No addition, subtraction  Compare value with 0 , update Z flag // to check that value is 0  load another value  call Endline  call Split value  value1 ^ value2  if equal, // whether sign bit is equal or not  call Addition function  else,  call Subtraction function  Split value // S: sign bit, E: exponent, M: mantissa  Store S of value1, E of value1, M of value1  Store S of value2, E of value2, M of value2  Normalize mantissa  Compare S of value1 with S of value2  If equal,  Call Addition function  Else  Call Subtraction function  Addition  Compare E of value1 with E of value2  If (E1 > E2)  R8 = E1 - E2  Else  R8 = E2 – E1    Normalize mantissa // to adjust bigger value  R9 = M1 + M2, update flag  If Carry set  Normalize mantissa and adjust Exponent  Else  Normalize mantissa and Store it into R9 again  Call Cal\_Result function  Subtraction  Compare E of value1 with E of value2  If (E1 > E2)  R8 = E1 - E2  Else  R8 = E2 – E1    Normalize mantissa // to adjust bigger value  R9 = M1 + M2, update flag  If Carry set  Normalize mantissa and adjust Exponent  Else  Normalize mantissa and Store it into R9 again  Shift mantissa  Normalize mantissa  While( N flag =0)  {  Call shift mantissa function  }  If mantissa is 1.xxx  Mantissa = Mantissa – 0x800000  R9 = Mantissa  Call cal\_result function  Cal\_result  Shift r11 to left as 31bit // sign bit  Store r11 into r10    Shift r3 to left as 23bit // exponent  R10 = r10 + r3  R10 = r10 + r9 // mantissa  Call Endline function    Endline  Program counter  } |

* 1. Flow chart 작성

|  |
| --- |
|  |
| 위 그림은 Main함수의 Flow chart이다. |
|  |
| 위 그림은 Split\_Value함수의 Flow chart이다. |
|  |
| 위 그림은 Addition함수의 Flow chart이다. |
|  |
| 위 그림은 Subtraction(+Shift\_mantissa) 함수의 Flow chart이다. |
|  |
| 위 그림은 Cal\_Result함수의 Flow chart이다 |

* 1. Result
     + 1. **Value2개중 1개가 0인 경우**

|  |  |  |
| --- | --- | --- |
|  | |  |
| 1) 위의 왼쪽 그림은 디버깅하자마자 register화면이다. 모든 값이 0으로 초기화 되어있음을 확인할 수 있다.  위의 오른쪽 그림은 r0에는 0XC14C0000 r1에는 0을 LOAD한뒤의 register화면이다.  정확하게 저장되어있음을 확인할 수 있다. | | |
|  |  | |
| 2) 왼쪽 그림은 R1이 0이기 때문에 조건 연산에 들어가서 Z flag가 set된 것을 확인할 수 있다. 따라서 바로 r0의값을 해당 주소 값에 저장하였고 위의 그림을 보면 r1이 0일 때 r0의 값인 0xC14C0000이 그대로 해당주소번지에 저장 되어있음을 확인할 수 있다. | |

* + - 1. **value2개가 같은 부호일 경우**

|  |  |  |
| --- | --- | --- |
|  |  | |
| 1) 위의 왼쪽 그림은 디버깅하자마자 register화면이다. 모든 값이 0으로 초기화 되어있음을 확인할 수 있다.  위의 오른쪽 그림은 r0에는 0x41240000(10.25), r1에는 0x4185000(16.6875)을 r12에는 지정된 주소 값을 LOAD한뒤의 register화면이다. 정확하게 저장되어 있음을 확인할 수 있다. | | |
|  |  | |
| 2)위의 왼쪽 그림은 Split\_Value함수를 호출하여 r2~r4까지 value1의 sign bit, exponent, 1.mentissa 값을 따로 저장하였고, r5~r7까지 value2의 sign bit, exponent, 1.mentissa값을 저장한 뒤의 register화면이다. 위의 오른쪽화면은 sign bit끼리 비교연산하여 같으므로 Z flag가 set되서 Addition함수를 호출하기전의 register화면이다.  비교 연산은 TST를 사용하였다. | | |
|  |  | |
| 3) 위 왼쪽 그림은 두 value의 exponent값을 비교(CMP) 하여 value2의 exponent가 더 커서 n flag가 set 됐음을 확인할 수 있고, 오른쪽 사진은 그에 따른 연산을 통해 value1와 value2의 mantissa값을 맞춰주었고 exponent를 맞춰주었다. r3, r6가 같아졌음을 확인할 수 있고, r4의 값이 변화되었음을 확인할 수 있다. | | |
|  |  | |
| 4) 위의 왼쪽 사진은 덧셈 연산을 위해 r4, r7을 최상위 비트까지 비트 이동을 한 뒤 두 mantissa값을 더한 뒤 flag를 update한뒤의 register화면이다. 따로 carry가 set이 되지 않았고, 오른쪽 사진을 보면 바로 다시 원래 mantissa해당 비트로 비트 이동을 한 뒤의 register화면이다. | | |
|  | |  |
|  |
| 5) 위의 왼쪽 사진은 r9를 다시 0.mentisa꼴로 맞추기 위하여 0x800000을 뺀 뒤의 register화면이다. 오른쪽 사진은 모든 연산이 끝나서 Cal\_Result를 호출한 뒤 sign bit과 exponent와 mantissa값을 r10에 저장한 뒤의 register화면이고 지정 메모리 주소 값 0x40000에 저장하였다. 10.25(41240000)+16.6875(41858000)=26.9375(41D78000)이다. 정확히 계산되어 저장되어 있음을 확인할 수 있다. | | |

* + - 1. **value 2개가 다른 부호일 경우**

|  |  |
| --- | --- |
|  |  |
| 1) 위의 왼쪽 그림은 디버깅하자마자 register화면이다. 모든 값이 0으로 초기화 되어있음을 확인할 수 있다.  위의 오른쪽 그림은 r0에는 0xC14C0000(-12.75), r1에는 0x4185000(16.6875)을 r12에는 지정된 주소 값을 LOAD한뒤의 register화면이다. 정확하게 저장되어 있음을 확인할 수 있다. | |
|  |  |
| 2)위의 왼쪽 그림은 Split\_Value함수를 호출하여 r2~r4까지 value1의 sign bit, exponent, 1.mentissa 값을 따로 저장하였고, r5~r7까지 value2의 sign bit, exponent, 1.mentissa값을 저장한 뒤의 register화면이다. 위의 오른쪽화면은 sign bit끼리 비교연산하여 같으므로 Z flag가 clear되서 Subtraction함수를 호출하기전의 register화면이다. 비교 연산은 TST를 사용하였다. | |
|  |  |
| 3)위의 왼쪽 화면은 Subtraction함수로 들어가서 연산을 할 수 있도록 각 value의 exponent와 mantissa값을 맞춰주고, mantissa값을 비교하여 큰 값에서 작은 값을 빼서 r9에 저장한 뒤의 register화면이다. 오른쪽 화면은 그 뒤 Shift\_mantissa함수로 들어가서 r9와 0x800000을 CMP한뒤의 register화면이다 N flag가 set된 것을 확인할 수 있다. N flag가 clear될때까지 r9를 왼쪽으로 비트 연산을 한번씩 해주면서 동시에 exponent를 1씩 빼준다. | |
|  |  |
| 4) 위 오른쪽 화면은 r9가 1.mantissa꼴이 되어 반복을 그만하고 0.mantissa꼴로 만들어주기 위하여 0x800000을빼준뒤의 register화면이다. 오른쪽화면은 그후 Cal\_Result를 호출한 뒤 sign bit와 exponent와 mantissa값을 r10에 저장한 뒤의 register화면이고 지정 메모리 주소 값 0x40000에 저장하였다. | |
|  | |
| 5) 위 사진은 결과값을 저장한 메모리 화면이다.  -12.75(C14C0000)+16.6875(41858000)=3.9375(407C0000)이다. 정확히 계산되어 저장되어 있음을 확인할 수 있다. | |

* 1. Performance

|  |  |
| --- | --- |
|  |  |
|  |  |
| 1. 위의 첫 줄의 그림을 보면 sign bit가 같은, 즉, 부호가 같은 2개의 value의 단순 덧셈 연산을 했을 때의performance이다.   Code size 는 288 bytes , States 는 67인 것을 확인할 수 있다. 따라서 Score = 288 \* 672 = 1,292,832 이다.   1. 위의 둘째 줄의 그림을 보면 sign bit가 다른, 즉, 부호가 다른 2개의 value의 뺄셈 연산을 했을 때의 performance이다. Code size 는 288 bytes , States 는 88인 것을 확인할 수 있다. 따라서 Score = 288 \* 882 = 2,230,272이다. | |

1. 고찰 및 결론
   1. 고찰 (35%)

이번 실습 과제는 floating point를 이해하고 이를 어셈블리 언어로 구현하여 floating point로 표현한 두 실수 간의 addition을 구하는 문제였다. 인터넷 강의를 통해 floating point의 개념을 미리 배우고 왔기 때문에 개념 자체를 이해하는 데에는 크게 어려움이 없었다. 이를 어셈블리언어로 표현하는 것 까지는 괜찮았지만 이를 이용하여 덧셈을 표현을 하려고 하니 굉장히 어려웠다.

가장 상위 bit는 sign bit 그 다음 8bit는 exponent 그 다음 23bit는 mantissa를 나타내었다. Sign bit과 exponent는 비트 범위가 정확히 정해져 있어서 계산하기 쉬웠지만 문제는 mantissa였다 23비트중 몇 개의 비트를 사용하는지 정확하지 않고 carry가 발생할 수 있기 때문이다. 이 부분을 계산하는 것이 굉장히 어려웠다. 여러가지 우여곡절을 통해 가장 상위 비트로 비트 연산을 한 뒤, 덧셈을 한 후, 다시 원래 비트 자리로 비트 연산을 하는 방식으로 구현을 하였다. 이때 Carry가 발생한다면 exponent의 값도 변경해주었다. 이 부분을 생각하는 것이 굉장히 어려웠다. 또한 두 value의 sign bit이 같을 때와 다를 때의 연산 방법이 달라서 이 부분을 처음엔 한꺼번에 표현을 하려고 했을 땐 계속 오류가 났고 이후 두 가지 case로 나누어서 구현을 하였다. 또한 0을 value로 받았을 때 따로 연산을 하지않고 나머지 값을 출력하는 식으로 구현하였다.

이렇게 긴 코드의 어셈블리 구현은 처음이었지만 이번 과제를 통해 좀더 어셈블리의 기본적인 문법들을 굉장히 많이 사용하였고 잘 기억이 나지않아 그전의 강의자료들을 다 찾아보면서 구현을 하였고, 조건연산이나 비트연산실력이 전보다 훨씬 향상된 느낌을 받았다.

* 1. 결론 (10%)

Floating point 덧셈연산에서 중요한 점은 두 value값이 같은 sign bit을 가지고 있나, 다른 sign bit을 가지고 있느냐가 가장 핵심이었다. Sign bit이 같다면 exponent를 맞춰준 후 해당 mantissa를 비교하여 더한 뒤 연산하고 carry발생시 exponent를 추가한 뒤 다시 비트 이동을 해주면 되었고, sign bit이 다를 경우에는 덧셈 결과의 sign bit이 두개의 value중 절대값이 큰 value의 부호를 따라간다는 점과, mantissa연산간의 exponent의 비트이동으로 인한 연산 후의 재 비트 이동과 동시에 exponent값의 변경 부분이 가장 중요한 부분이었다. 이 부분은 1.mantissa의 형식으로 만들어주기 위하여 exponent를 1씩 감소하며, 왼쪽으로 mantissa를 비트 이동한 뒤 해당 형식을 갖추었으면 다시 0.mantissa의 형식으로 맞추어 저장하는 식으로 구현하였다.

Floating point addition을 구현하였기 때문에 이 프로그램을 통해서 앞으로는 정수의 사칙 연산만이 아닌 실수의 사칙연산도 구현이 가능할 것으로 생각이 든다.

1. 참고문헌(2%)

강의자료

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