어셈블리 프로그래밍 설계 및 실습

실험제목: Pseudo Instructions

실험일자: 2017년 11월 02일 (목)

제출일자: 2017년 11월 08일 (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

Pseudo Instructions

* 1. 목적

Pseudo Instruction 에 대해 이해하고 그 특징과 장점을 파악하여 어셈블리 프로그래밍에 응용해 본다. 디버깅 시 Disassembly를 확인하여 Pseudo Instruction이 실제로 어떤 Instruction으로 변환되는지 확인해보고, Memory 나 Register 값을 확인해 보면서 Label이 의미하는 것이 주소영역임을 이해한다.

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

|  |
| --- |
| Main  {  CharCpy =0;  Table1= “Hello World”  Table2= NULL  Copy(Table1, Table2)  }  Copy  {  Stack.push(R0-R5)  J = 0;  For( int i=0; Tabel1[i]!=NULL ; i++)  {  If( Table1[i] == 0x0D)  Break;  If( Table1[i] == “ ”)  Continue;  Table2[j] = Table1 [i]  J++;  R6= R6+1  }    Stack.pop(R0-R5)    Return R6  } |

* 1. Flow chart 작성

|  |
| --- |
| C:\Users\장한별 (Stanley)\Desktop\flowchart.jpg |

위 그림은 Problem 의 Flow chart 이다.

* 1. Result

1. Register 및 Memory

|  |  |  |
| --- | --- | --- |
|  |  | |
| 1) 위 그림은 처음 디버깅 했을 때의 register화면이다. 모든 레지스터의 값이 0으로  초기화 되어있음을 확인할 수 있고, 오른쪽 그림을 보면 0x51부터 Table1에 저장하였던 “Hello World”, cr 이 1byte단위로 저장되어있음을 확인할 수 있다.  cr을 0x0d로 설정해놓았기 때문에 가장 마지막에 0x0d가 저장되어있음을 확인할 수 있다. | | |
|  |  | |
|  | |
| 2) 위 왼쪽 그림은 r0, r1, r2에 Table1, Table2, CharCpy의 주소 값을 차례로 저장하고 함수를 호출하기 직전의 register화면이다. lr에 다음 pc값을 저장하였다.  위 오른쪽 그림은 함수가 호출 된 뒤 stack에 Full ascending방법으로 r0부터 r5의값을 저장시킨 뒤의 메모리 화면이다. 정확하게 저장되어있음을 확인할 수 있다. | | |
|  | |  |
| 3) 위 그림은 Copy함수가 호출 된 뒤 Table1의 문자열을 Table2에 다 복사한 뒤의 register화면 및 memory화면이다. 보면 0x51이부터 0x5C까지 Table1의 값이 저장되어 있고, 0x60부터 0x69까지 Table1의 값이 복사되어 Table2의 주소 값에 저장되있음을 확인할 수 있다. 이때, 0x56에 20은 복사되지 않았는데 이는 띄어쓰기로 저장을 시키지 않았기 때문이다. 또한 0x50에는 배열된 문자열의 사이즈인 10이 저장되어 있음을 확인할 수 있다. 오른쪽 사진은 초기에 스택에 저장되었던 값들을 함수가 끝나고 다시 pop하여 초기값으로 한번에 초기화한 Register 화면임을 확인할 수 있다. | | |

1. Disassembly

|  |
| --- |
| C:\Users\장한별 (Stanley)\Desktop\dis1.JPG |

위 그림은 전체 코드의 Disassembly 화면이다.

C:\Users\장한별 (Stanley)\Desktop\d1.JPG

위 그림은 LDR r0, =Table1 명령어의 disassembly 코드로 나타낸 화면이다. PC에서 0x003C만큼 떨어진 곳의 값을 R0에 넣는다고 해석이 되어있다. 또한 32bit의 코드를 보면 E59F003C로 나와있는데, 이를 해석하면 가장 상위 4bit는 condition Field로서, E(1110)이 써져 있는 것을 볼 수 있다. 어떠한 조건도 없기 때문에 AL 즉 항상 수행한다는 뜻으로 해석된다.

다음 4bit를보면 5로 되어있고, 이는 25번째 비트가 0, 24번째 비트가 1의 뜻이다. 25번째비트는 0의 뜻은 data를 transfer할 때 12bit의 상수 값을 인자로 받는다는 뜻이고, 24번째 비트가 1인 것은 pre-index의 방식을 적용하겠다는 의미이다. 그 다음 4bit는 9로서, 1001로 표현 가능한데, 이는 UP의방식, word단위, write back방식사용x, load를 사용한다는 의미이다.

그 다음4bit는 F(1111)으로 base 가 1111 즉 R15(PC)이라는 뜻이고, 그 다음 4bit는 source /destination register인데 이는 0으로써 없다는 뜻이다. 그 다음 12bit는 offset으로서, 위 경우에는 12bit의 상수 값을 받게 되고 0x3C가 저장 되어 있음을 확인할 수 있다.

C:\Users\장한별 (Stanley)\Desktop\d2.JPG

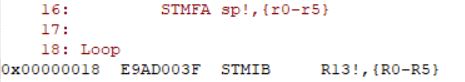
위 그림은 MOV lr, pc 명령어의 disassembly 코드로 나타낸 화면이다. R14에 PC의값을 넣는다고 해석 되어있다. 또한 32bit의 코드를 보면 E1A0E00F로 나와있는데, 이를 해석하면 가장 상위 4bit는 condition Field로서, E(1110)이 써져 있는 것을 볼 수 있다. 어떠한 조건도 없기 때문에 AL 즉 항상 수행한다는 뜻으로 해석된다.

그 다음8bit을 한번에 보게 되면 1A로서, 00011010으로 표현되고, 25bit가 0인 것은 operand2의 사용방법을 나타내는 것인데, 4bit가 1이므로, 레지스터 값을 사용한다고 볼 수 있다. 또한 21bit부터 24bit를보면 1101임을 확인할 수 있는데 이는 MOV Mnemonic임을 확인할 수 있다. 이후 16bit부터 19bit를보면 0000으로 first operand register는 없고, 이후 4bits는 destination register로서, 1110 즉 r14를 나타내는 것으로 확인할 수 있다. 이후 operand2는 000000001111으로서, second operand register가 1111 즉 R15(PC)를 나타내는 것 확인할 수 있다

C:\Users\장한별 (Stanley)\Desktop\d3.JPG

위 그림은 B Copy 명령어를 disassembly 코드로 나타낸 화면이다. 0x18로 branch한다고 해석이 되어있다.

또한 32bit 코드를 보면 EA000000로 나와있는데 이를 해석하면 가장 상위 4bit는 1110으로서 condition field로, AL 즉 조건 없이 항상 수행한다는 뜻으로 해석된다. 그 다음 24bit를보면 이게 1로 set되면 BL이고 0으로 clear이면 그냥 B라는 뜻인데 BL이아닌 B 를 했으므로 0으로 clear됬음을 확인할 수 있다. 그 이후 24bit는 signed word offset이다.



위 그림은 STMFA sp!, {r0-r5} 명령어를 disassembly코드로 나타낸 화면이다. R13(stack pointer) 에, r0부터 r5의 값을 Full ascending으로 저장한다고 해석된다. 또한 32bit의 코드를 보면, E9AD003F로 나와있고, 어떠한 조건이 없으므로, 맨 앞 4bit의 condition field가 1110으로 항상 수행한다고 되어있다.

그 다음 25bit는 1로서, pre-index방법을 사용했다고 해석되고, 그 다음 4bit를 보게 되면 1010 로서 UP방식, restore PSR and force user bit x, write back o, store방식을 사용한 것을 확인할 수 있다. 또한 그 다음 4bit는 base register를 나타내는데 1101이므로 R13임음 확인할 수 있다. 그 다음 16비트는 register의 리스트이다.

C:\Users\장한별 (Stanley)\Desktop\d5.JPG

위 그림은 CMP r3, #cr 명령어를 disassembly 코드로 나타낸 화면이다. R3와 0x0000000D를 CMP한다고 해석된다. CR을 0x0D로 Define했기 때문에 이렇게 보여진다. 32bit 코드를 보면, E353000D로 나와있고, 어떠한 조건이 없으므로 맨 앞 4bit의 condition field가 1110으로 항상 수행한다고 되어있고, 25bit는 1로서 operand2가 상수 값을 사용한다고 해석된다.

또한 21bit부터 25bit는 1010으로서, opcode를 나타내는데, CMP를 나타낸 것을 확인할 수 있다. 20bit는 condition code를 set한다는 의미로 설정하라는 것을 확인할 수 있고, 그 다음 4bit는 first operand register로서, 0011 즉 R3를 의미하는 것으로 확인할 수 있다. 그 다음 4bit는 destination register로서, 0이므로 없는 것을 확인할 수 있고, 0bit부터 12bit 까지는 operand2를 나타내는데 0xD임을 확인할 수 있다.

C:\Users\장한별 (Stanley)\Desktop\d6.JPG

위 그림은 ADD r6, r6, #1 명령어를 disassembly 코드로 나타낸 화면이다. R6와 0x00000001을 더해서 R6에 저장한다고 해석된다. 32bit 코드를 보면, 어떠한 조건이 없으므로, condition field가 1110으로 항상 수행한다고 되어있고, 25bit는 1로 set되어 operand2가 상수 값으로 사용된다고 해석된다.

또한 21bit부터 25bit는 0100으로서, opcode를 의미하는데 ADD opcode임을 확인할 수 있다. 또한 20bit는 set condition code로서 clear됨을 확인할 수 있고, 그 다음 4bit는 first operand register로서, 0110 즉 R6를 뜻한다고 확인되고, 그 다음 4bit는 destination register로서, 0110 즉 R6를 나타낸다. 그 다음 12bit는 operand2로서, 1을 나타낸 것으로 확인할 수 있다.

* 1. Performance

|  |  |
| --- | --- |
| C:\Users\장한별 (Stanley)\Desktop\st.JPG | C:\Users\장한별 (Stanley)\Desktop\co.JPG |

위의 그림은 Problem 의 performance이다. Code size 는 100 bytes, States 는 195인 것을 확인할 수 있다. 따라서 Score = 100 \* 1952 = 3,802,500이다.

1. 고찰 및 결론
   1. 고찰 (35%)

이번 실습은 특정 문자열을 받아와 그 문자열을 다른 주소 값에 저장하는, C언어에서의 strcpy의 기능을 하는 함수를 어셈블리 언어로 구현하는 문제였다. 이때, 문자열의 끝을 확인하기 위하여 cr을 문자열의 마지막에 저장하였고, 이는 0x0D의 값을 갖도록 설정하였다.

실습 시간에 특정 문자열의 크기를 구하는 strlen 코드를 작성하였고, 이를 잘 구현하고 설명을 들어서 실습 문제인 strcpy의 코드를 짜는 데에는 크게 문제가 없었다. 하지만 문제가 있던 곳은 R/W Base에 있었다. 실습 시간 예제 코드를 작성하는데 오류가 나서 이 부분을 R/W Base의 범위를 다시 지정해주면 오류가 나지 않는다는 설명을 들었지만 이 부분을 정확히 이해할 수 없었다. 필자는 강의 자료에 나와있는 0x0D로 모두 바꿔주면 되는 줄 알았지만 그게 아니었다. 코드가 바뀔수록 R/W Base의 범위도 바꿔줘야 하였다.

처음 배열 문자 값이 저장 주소에 나타나지 않아 힘들었다. 그 뒤 많은 시간에 걸쳐 모든 메모리 값을 확인하였고, R/W Base의 값을 계속 0으로 바꿔주면서 시작 범위를 확인하였다. 이 부분을 코딩을 진행하면서 이해하였고, 코드를 모두 작성한 뒤에 최종적으로 R/W Base의 시작 범위를 다시 확인하여 지정해주었고, 정확한 값을 도출할 수 있었다.

또한 disassembly의 언어의 대해서도 이해 할 수 있었다. 실습시간에는 딱히 배우지않았고, 설계 시간에 중점적으로 배웠던 모든 어셈블리 명령어는 32bit의 특정 값을 가지고 있는데, 설계 시간에 수업을 들었을 때는 이론적으로 배우기만 하니까 받아들이기가 쉽지가 않았다. 하지만 내가 직접 작성한 코드의 disassembly언어의 32bit코드가 어떤 방식인지 확인할 수 있었다.

* 1. 결론 (10%)

이번 실습시간을 통해 사람들이 쉽게 이해할 수 있는 pseudo instruction을 assembly언어로 옮겨 코딩 할 수 있는 능력을 기를 수 있었다. 또한 문자 배열이나 특정 값을 임의의 주소에 저장하면 이 부분을 디버깅을 통해서 어디에 저장되어 있는지 주소 값도 확인할 수 있었다. 코딩을 하는데 있어서, R/W Base의 중요성을 알게 되었다. 코드가 변함에 따라서 R/W BASE의 범위도 바뀌면서 재설정을 해주어야 원하는 값을 정확히 도출, 확인 할 수 있었다.

또한 Pseudo instruction이 어떻게 변경되고, 변경된 instruction이 32bit코드대로 프로그램에서 어떻게 동작하는지에 대해서 확인할 수 있었고, disassembly화면을 확인하고 이해할 수 있는 능력도 기를 수 있는 시간이었다.

1. 참고문헌(2%)

강의자료

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