어셈블리프로그래밍설계및실습 보고서

실험제목: Pseudo instruction

실험일자: 2016년 11월 03일 (목)

제출일자: 2016년 11월 16일 (목)

학 과: 컴퓨터공학과

담당교수: 이형근 교수님

실습분반: 화5, 목6,7

학 번: 2013722095

성 명: 최 재 은

1. 제목 및 목적
   1. 제목

Pseudo instruction

* 1. 목적

Pseudo instruction의 특징과 장점을 알아보고 이를 통해 지정된 데이터 영역의 특정 변수로 쉽게 접근하는 방법을 이해한다. 또한 disassembly를 보고 분석하여 Pseudo Instruction이 실제로 어떤 assembly instruction으로 변환되는지를 알고 원리를 이해한다.

이를 위해 Pseudo code를 보고 이를 분석하여, 기계어로 프로그래밍 하는 능력을 배양한다.

1. 설계 (Design)
   1. Pseudo code

CR을 0x0d로 정의

R0에 Arr1의 주소, R1에 Arr2의 주소를 저장, R7에 loop lable의 주소저장

k주소 번지의 값을 1로 초기화

현 위치를 lr에 저장하고, copy\_arr\_wo\_space함수 호출

R5 값을 r2주소번지에 저장

- **copy\_arr\_wo\_space**

R0 ~ R5까지 stack에 저장

R4(I), R5(J)를 0으로 초기화

**- loop**

R0+r4(index)주소번지의 값을 r3에 저장

R3가 CR과 같으면 done lable로 이동

R3가 0x20(space)와 같으면 r5(저장위치 감소를 위해 빼줄 값)을 1증가, r4도 1증가하고 pc를 R7이 가진 주소 값 위치로 이동

R4 - R5값을 r6에 저장( i - j )

R1+R6 주소 값 위치에 r3를 저장(arr1의 한 글자를 arr2에 복사)

R4(index)와 r9(arr2의 index)를 1증가

pc를 R7이 가진 주소 값 위치로 이동

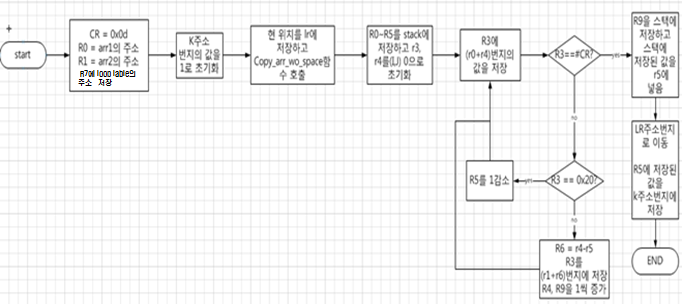
**- DONE**

Sp가 가리키는 주소번지에 r9을 저장(push)

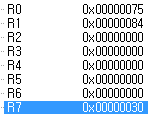
Sp가 가리키는 번지의 값을 r5저장(pop)

Lr 주소번지로 이동

* 1. Flow chart 작성



* 1. Result



- r0에 arr1의 주소, r1에 arr2의 주소, r7에 loop lable의 주소가 저장되었습니다.

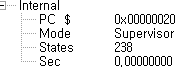


- r0, r1의 주소가 stack에 저장되었습니다.



-위의 사진을 보면 0x74에 space를 제거하여 복사를 한 arr2의 길이 12가 저장이 되어있음을 확인할 수 있고, 큰 box를 보면 정상적으로 arr2에 복사가 되었음을 확인할 수 있습니다.

* 1. Performance





- code size는 136이고 state는 238개입니다. Score는 code size \* state^2이므로

이 프로그램의 스코어는 7703584가 나오게 됩니다.

1. 고찰 및 결론
   1. 고찰 및 결론

**1) 코드 비교**

- 오른쪽 사진이 슈도 코드에서 arr1과 arr2, k를 선언하고

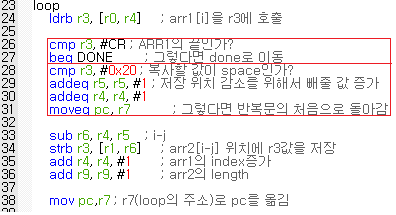
초기화하는 부분입니다.

- 왼쪽 사진이 k를 1로 초기화한 것입니다.



- 오른쪽을 보면 r0~r5 stack에 push하고

I인 R4와 j인 R5를 0으로 초기화합니다.



- 왼쪽 사진의 첫 번째 박스가

If(arr1[i] is cr) break; 에 대한 조건문이고 두 번째 박스가

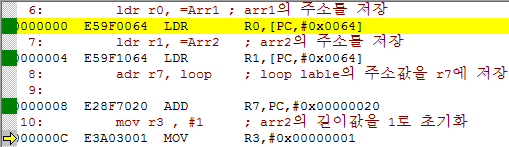
If(arr1[i] is space) incr j; continue;

에 대한 조건문입니다.

그 밑이 arr1[i]를 arr[i-j]에 대한 코드입니다. 두 조건문에 해당되지 않으면 loop의 안에서 반복합니다.

- 오른쪽의 사진이 반복문을 탈출하고 나서 arr2의 길이를 stack에 push하고 pop하는 코드입니다.

**2) Disassembly code**



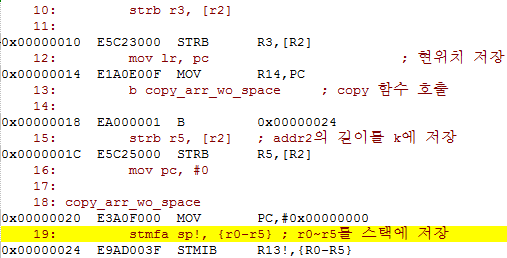
- LDR과 ADR의 operand는 실재로는 Rd, [pc, #target\_address-8-pc]로 동작한다.

(pipeline 동작 때문에 실 주소위치는 -8을 해주어야 함)

- Arr1의 address가 저장될 위치는 0x6c이므로 arr1, arr2주소값이 들어갈 자리 8을 뺀 0x0064로 표기됨. Arr2의 address 또한 위와 같은 방식으로 저장이 표기됨.

- loop lable의 주소는 0x2c인데, 2c(48)-8-8 = 0x20(32)이므로 r7, pc, #0x00000020으로 표기되며, 이때 pc와 이를 더한 값을 R7에 넣어준다.

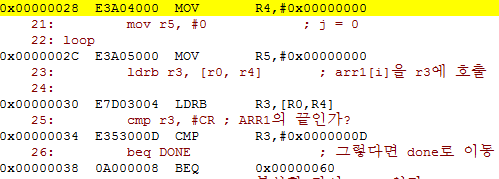
- mov instruction은 기존과 동일.



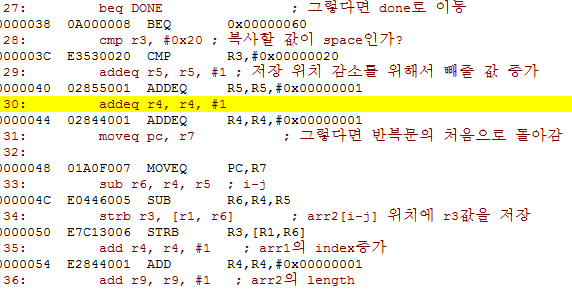
- STRB 또한 동일하며, copy\_arr\_wo\_space lable의 주소가 0x00000024이므로

b copy\_arr\_wo\_space라고 입력하면 실재로는 b 0x00000024가 실행됨.

- STMFA 는 실재로는 Increment Before 동작을 수행하므로 IB Addressing mode로 STM이 실행됨.



- beq DONE 코드 또한 copy\_arr\_wo\_space과 동일하게 실재로는 DONE lable의 주소로 점프하는 동작이 수행됨.



- R3가 0x20인가 비교하는 코드 또한 CR처럼 선언하고 사용할 수 있지만 그냥 0x20으로

cmp하였음. 동작은 동일할 것이라 판단됨.

- moveq pc동작 또한 그대로 pc가 r7의 주소를 받아 이동함.

- add, sub instruction은 기존의 코드와 동일한 동작 수행

**3) Instruction’s structure**



- 명령어의 형태는 위와 같이 cond[31:28], opcode[24:21](arithmetic/logic function)

S(set condition code), Rn[19:16](first operand register), Rd[15:12](destination register)

Operand2[11:0]으로 구성되어있습니다.



- CMP R3, #CR은 1110/00/1/1010/1/0011/0000/0000/00001101으로 구분이 되며 Rn이 3번이고, immediate alignment =0000, opcode는 GE, set condition =1이라고 볼 수 있으며 이를 해석하면 3번 레지스터와 operand2의 후위 8bit와의 CMP(비교)라고 볼 수 있습니다.



- MOV R3, #1은 1110/00/1/1101/0/0000/0011/0000/00000001로 구분할 수 있으며

Rd는 3번 레지스터, #이 1이므로 immediate alignment = 0000이고, opcode = 1101,

S =0으로 보아 R3에 operand2의 후위 8bit 00000001을 mov(opcode)하겠다는 것으로 해석할 수 있습니다.



- MOVEQ pc, r7은 0000/00/0/1101/0/0000/1111/000000000111로 구분할 수 있으며 R15(Program counter)가 Rd이고, #이 0이므로 operand2를 00000/00/0/0111로 구분지어 Rm이 7번 레지스터가 되고 오른쪽 방향으로 00000만큼 shift하므로 이를 해석하면

R7을 shift하지 않고 R15에 mov(opcode)하겠다가 됩니다.



-ADR은 실재로 ADD의 동작을 수행하는데 ADR R7, loop는

1110/00/1/0100/0/1111/0111/000000100000으로 구분할 수 있으며 Rd = 7 Rn =15

#이 1이므로 immediate alignment = 0000, operand2는 사실상 0x20입니다. Opcode가 0100이므로 이를 해석하면 R15(PC)와 0x20을 더해서 R7에 저장한다가 됩니다.



-ADDEQ r5, r5, #1은 0000/00/1/0100/0/0101/0101/000000000001로 구분이 되며 Rd = 5, Rn = 5, #이 1이므로 immediate alignment = 0000이고 operand2는 사실상 0x01로 볼 수

있습니다. Opcode가 0100이므로 이를 해석하면 R5에 0x01을 더해서 R5에 저장한다가 됩니다.



-SUB R6, R4, R5는 1110/00/0/0010/0/0100/0110/000000000101로 구분되어, Rd = 6,

Rn =4, #이 0이므로 operand2를 00000/00/0/0101로 구분하며 Rm = 5가 됩니다. 이 때

Opcode는 0010이므로 이를 해석하면 R4에서 R5를 뺀 값을 R6에 저장한다가 됩니다.





- LDR r0, =Arr1의 경우 1110/01/0/1/1/0/0/1/1111/0000/000001100100로 구분할 수 있으며 LDR r1, =Arr2의 경우 1110/01/0/1/1/0/0/1/1111/0001/000001100100로 구분되어 뒤에서 네 번째 숫자는 레지스터 번호임을 알 수 있습니다. #이 0이고 p가 1므로 offset을 pre-index로 사용하며, U =1이므로 up 방향으로 적용되며, unsigned byte로 인식합니다.

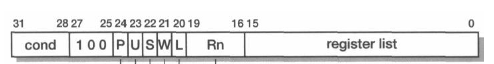
W=0이므로 거꾸로 쓰지 않으며, L=1이므로 load 동작을 수행합니다.



- STRB r3, [r1, r6]는 1110/01/1/1/1/1/0/0/0001/0011/000000000110으로 구분하여 Rd = 3

Rn =1, #이 1이므로 offset을 0000/00/0/0110으로 구분하여 Rm(offset register) = 6이고 Shift하지 않습니다. L=0이므로 store를 하고, w=0이므로 write-back하지 않으며 B =1이므로 unsigned byte로 기록되며, U=1이므로 UP방향으로, p=1이므로 pre-index를 합니다.

즉, R1에 R6를 pre-index로 더한 주소 위치의 값을 r3에 unsigned byte로 store하겠다가 됩니다.





-STMFA sp!, {r0-r5}는 1110/100/1/1/0/1/0/1101/0000000000111111로 구분되며

Base register Rn = 13(stack pointer)이고 P=1(pre-index), U=1(up), S=0,W=1(write-back),L=0(store)이고 register list code를 보면 0~5번째 bit가 1이다.

이를 해석해 보면 R0~R5를 SP를 기준으로 하여 pre-index로 write back할것이며 이들은 위쪽방향(주소값이 커지는 방향)으로 기록할 것이다가 된다.





- b copy\_arr\_wo\_space는 b 0x00000024(copy\_arr\_wo\_space lable의 주소)이며

이는 1110/101/0/000000000000000000000001으로 구분할 수 있는데 이를 분석해보면

L =0이므로 기본 Branch를 사용하겠다는 것이므로 0x24주소 위치로 pc를 이동시키겠다는 것으로 해석할 수 있습니다.

**4) 고찰 및 결론**

이번 과제를 하면서 가장 큰 의문은 왜 레지스터들을 ‘왜 스택에 저장하는가’였습니다. 어차피 이 코드의 목적은 배열을 복사하고 복사된 배열의 길이를 반환하는 것인데, ‘stack에 넣었다 뺐다 하는 일련의 과정이 오히려 state와 code size를 늘리는 것이 되지 않나?’라는 생각이 들었지만, 실제로 큰 사이즈의 프로그램을 짜면서 이번 실험의 코드를 sub function으로 사용하게 되면 ‘이전에 레지스터에 있던 값을 날려먹을 순 없으니 스택에 저장하는구나’라는 생각이 들었습니다. **무릎을 탁 쳤습니다**. 과제를 진행하면서 처음에 레지스터 0~5까지에 문자열을 받아서 스택에 저장하고 그걸 byte단위로 읽어 들여서 비교하는 식으로 구현하려다가 위의 무릎을 탁 치는 깨달음을 얻고는 ‘아 이래서 lab에 슈도 코드가 그렇게 적혀있었구나’했습니다. 그전에는 그냥 strcpy를 구현하는 데에만 급급했었거든요….아무튼 잘 구현되었습니다. 코드를 짜면서 큰 어려움은 없었지만 instruction code를 분석하는 부분이 꽤 시간을 잡아먹지 않았나 합니다.

1. 참고문헌

[https://ko.wikipedia.org](https://ko.wikipedia.org/wiki/%EC%9C%84%ED%82%A4%EB%B0%B1%EA%B3%BC:%EB%8C%80%EB%AC%B8)

<http://blog.daum.net/tlos6733/132>